程序设计思想与方法

wizardforcel

Published with GitBook

目錄

程序设计思想与方法

陆朝俊 编著

前言

2006 年 3 月，美国计算机科学家 Jeannette M. Wing（周以真）在 CACM 上发表文章《计算思维》（Computational Thinking），主张计算机科学家应该向大学新生讲授一门关于如何“像 计算机科学家那样思考”的课程，这门课并非仅为计算机科学专业学生开设，更重要的是面 向所有非计算机专业的学生，甚至是面向中小学学生。进行计算思维教学的目标是使计算思 维像阅读、写字、算术一样成为每个人的基本技能。

所谓“计算思维”，是指运用计算机科学的基础概念、思想和方法去解决问题时的思维活 动，涉及如何在计算机中表示问题、如何让计算机通过执行有效的算法过程来解决问题。计 算机原本只是人们解决问题的工具，但当这种工具在几乎每一个领域中都得到广泛使用后， 工具就会反过来影响人们的思维方式。因此，将计算思维向所有人进行普及，使普通人群也 能像计算机科学家那样利用计算机来解决自己生活、工作中的问题，对于人们适应未来的、 计算机无处不在的社会，具有重要意义。

上海交通大学为全校学生开设一门称为“程序设计思想与方法”的通识课程已有多年， 从 2010 学年秋季学期开始，我们对该课程进行改革，试图将它转变成计算思维课程。由于计 算思维是一门崭新的课程，国内外都没有合适的教材，甚至计算思维课程应当讲授的内容也 没有定论，这促使我们按照自己的理解编写了这本计算思维教材。

目标

本书向计算机专业和非计算机专业的学生介绍计算机科学的基本概念、思想和方法，目 的是使学生理解计算机科学家的思维特点和方式，并最终能够利用计算机解决自己专业领域 的问题。

内容

本书内容覆盖利用计算机解决问题的全过程。

第 1 章首先界定“计算”的含义，然后介绍“计算思维”的基本内容。计算是指利用计 算机解决问题的过程，而非传统意义的数学计算，其实质是“算法化”，即按照一定的步骤执 行基本指令的过程。为了让学生实践所学到的计算机问题求解的思想和方法，需要利用某种 编程语言来实现算法，本书采用 Python 语言作为编程的教学工具。

计算机可以看作是信息处理机器，所有问题的解决都是对特定信息进行特定处理的过程。 第 2 章和第 6 章介绍如何在计算机中表示现实世界信息，其中第 2 章介绍简单信息的表示，包括数值、字符串等；第 6 章介绍复杂信息的表示，包括各种集合体数据和数据结构。

第 3 章介绍如何表示对信息的处理过程，包括顺序、条件、循环等控制流程的表示以及结构化编程的思想。第 4 章介绍如何将信息处理过程按照良好的结构组织起来，模块化编程 和自顶向下设计可以帮助我们建立复杂问题的处理过程。

第 5 章介绍图形编程。图形是传达信息的最高效的手段，在利用计算机解决问题时经常 用图形来实现可视化计算，因此图形编程的重要性不言而喻。同时，早早地让学生学会图形 编程并编制一些有意思的程序，能够激发他们的学习兴趣。第 8 章介绍的图形用户界面是图 形编程的进一步延伸。

第 7 章详细介绍当前流行的面向对象编程。面向对象不只是一种编程范型，它还是一种 强大的思维工具，可以说是本书的重点内容。

传统计算都是确定性的，第 9 章介绍两个与不确定性打交道的内容。计算机模拟是在各 行各业中广泛应用的方法，本章介绍如何利用蒙特卡洛方法模拟现实世界中的不确定性。另 外，本章还简单介绍了多线程并发计算。

第 10 章介绍算法设计和分析。这章内容涉及理论计算机科学，旨在使读者了解计算机的 计算能力和局限。

第 11 章介绍所谓“计算+X”，说明计算机与各专业的结合能够形成多种交叉学科，同时 也证明了计算思维课程的重要意义。笔者不可能了解各专业的知识，所以本章只能是浅尝辄 止。

为什么选用 Python

由于计算思维课程要面向广大的非计算机专业学生，我们希望他们能够轻松地学会一种 编程语言，以便动手实践随后学到的知识。Python 语言非常简单，易学易用，可以让学生在 第一堂课就学会编写简单程序。另外，我们希望能用直观形象的方式来展开课程教学，Python 语言正好能满足我们的需求。Python 是一种编译/解释混合的高级编程语言，这使我们在课堂上可以通过会话方式来与 Python 解释器进行交互，即时演示教学内容。最后，Python 语言支 持我们希望在本课程中介绍的各种特性，如结构化编程、面向对象编程、图形和 GUI 编程多 线程等等，它完全可以用于开发实际的应用程序。

要说明的是，尽管本书介绍了很多 Python 语言的知识，但本书并不是“Python 语言”教 材，没有像编程语言教材那样介绍 Python。更多关于 Python 语言的内容，请参考专门的资料。

教学建议

首先，本教材包含的内容适合各专业学生的学习。对于非计算机专业的学生，可以忽略 那些较为深入的、涉及更多技术细节的内容，本书为这样的内容加上了“*”标记。

其次，在课堂上演示所教内容对于非计算机专业学生来说具有良好的效果，本书在编写 时充分考虑了这一点。在书中，有许多以下列形式出现的代码：

```
>>> print "Hello, World!"
Hello, World!
```

其中特意保留了 Python 解释器提示符“>>>”（并不是自己输入的），以提醒教师这样的代码可以当场演示。当然，任何阅读本书的读者都可以模仿这样的代码，边读书边动手实践。

致谢

上海交大计算机系有许多教师从事《程序设计思想与方法》课程的教学，笔者在与他们 的讨论、交流中获益匪浅。尤其是本课程改革的牵头人黄林鹏师兄，向笔者提供了很多资料、 建议和外校同行们的做法，非常感谢他的帮助。

感谢来自各专业的学生，他们在课堂内外的表现和提问，使笔者获得了向非计算机专业 学生讲授计算思维课程的经验。而很多学生在期末大作业中利用所学知识解决自己专业问题， 也令笔者很欣慰，说明本课程确实达到了目的。

为了了解本书是否适合非计算机专业的专业人士阅读，笔者请好友杨耀志、王爱琴伉俪 阅读了部分内容，非常感谢他们的反馈意见。

最后要感谢妻子和女儿的支持，忙碌的写作使笔者有些忽略了对她们的照顾。 由于作者水平有限，书中错误一定不少，恳请读者不吝赐教！

第 1 章 计算与计算思维

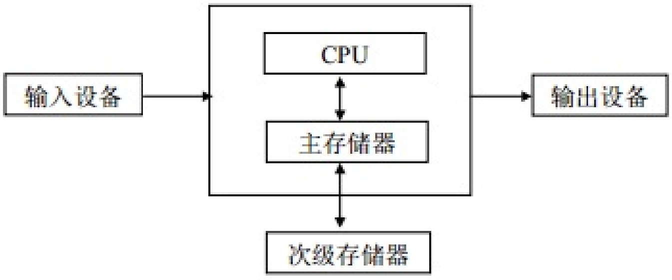
计算是利用计算机解决问题的过程，计算机科学是关于计算的学问。计算机科学家在用 计算机解决问题时形成了特有的思维方式和解决方法，即计算思维。本章介绍计算的基本概 念和计算思维的基本内容，而本书的其余章节将围绕计算与计算思维这个中心展开详细讨论。

1.1 什么是计算？

1.1.1 计算机与计算

计算机是当代最伟大的发明之一。自从人类制造出第一台电子数字计算机，迄今已近 70 年。经过这么多年的发展，现在计算机已经应用到社会、生活的几乎每一个方面。人们用计 算机上网冲浪、写文章、打游戏或听歌看电影，机构用计算机管理企业、设计制造产品或从 事电子商务，大量机器被计算机控制，手机与电脑之间的差别越来越分不清，……总之计算 机似乎无处不在、无所不能。那么，计算机究竟是如何做到这一切的呢？为了回答这个问题， 需要了解计算机的工作原理。

提到计算机，人们头脑中会首先浮现出显示器、键盘、主机箱等一堆设备——计算机硬 件。了解一点硬件设备的基本知识自然是需要的，不过从学习用计算机解决问题这个层次看， 并不需要掌握多少底层硬件知识。在此我们仅介绍现代计算机的主要功能部件，目的是要了 解用计算机解决问题的计算机制。现代计算机的主要功能部件如图 1.1 所示。



CPU、指令与程序 中央处理单元（CPU）是计算机的计算部件，能够执行机器指令，或简称指令（instruction）。

每条指令表达的是对特定数据执行特定操作。某种 CPU 能执行的全体指令是由该 CPU 的制造商设计并保持固定不变的，称为该 CPU 的指令集。例如，Intel 公司为它的 80x86 系列处理器设计了上百条的指令。

外行人也许会以为，计算机功能如此强大，必定是因为它能执行功能强大的指令。然而 事实并非如此。即使是当今技术最先进、计算能力最强大的计算机，它的 CPU 也只会执行一 些非常简单的指令，例如将两个数相加、测试两个数是否相等、把数据放入指定的存储单元 等等。

由于每条机器指令都只能完成很简单的操作，因此仅靠少数几条指令是做不了什么复杂 的事情的。但是，如果将成千上万条简单指令组合起来，却能解决非常复杂的问题！亦即， 复杂操作可以通过执行按特定次序排列的许多简单操作而实现。这种由许多指令按次序排列 而成并交给计算机逐条执行的指令序列称为程序（program）。为了用计算机解决问题，把问 题的解法表达成一个指令序列（即程序）的过程，称为程序设计或编程（programming）。可见，计算机所做的所有神奇的事情，都是靠一步一步执行的、平凡而乏味的简单指令序列做到的。计算机一点也不神奇，它唯一会做的事情就是机械地执行预定的指令序列。

存储器

存储器是计算机的记忆部件，用于存储数据和程序。 存储器分为主存储器和次级存储器，它们是用不同的物理材料制造的。CPU 只能直接访

问主存储器，也只有主存储器才能提供与 CPU 相匹配的存取速度。但主存储器需要靠持续供电来维持存储，一旦断电，存储的数据或程序就会消失。为了持久存储信息，可以使用即使断电也能维持存储的次级存储器，如当前普遍使用的磁盘。CPU 不能直接访问次级存储器，次级存储器上的数据或程序必须先送到主存储器中，才能被 CPU 存取或执行。次级存储器的读写速度远远低于主存储器，这个差别极大地影响了用计算机解决问题时所使用的方法。

现代计算机在体系结构上的特点是：数据和程序都存储在主存储器中，CPU 通过访问主 存储器来取得待执行的指令和待处理的数据。这称为冯·诺伊曼（von Neumann）体系结构。

输入/输出设备

输入和输出设备提供了人与计算机进行交互的手段。我们通过输入设备向计算机输入信 息，计算机则通过输出设备将计算结果告诉我们。传统的输入设备有键盘和鼠标等，输出设 备有显示器和打印机等。现代的触摸屏则兼具输入和输出的功能。

计算

了解了计算机的组成，就能理解计算机解决问题的过程是怎样的。我们来看一个常见任 务——用计算机写文章——是如何解决的。为了解决这个问题，首先需要编写具有输入、编辑、保存文章等功能的程序，例如微软公司的程序员们所写的 Word 程序。如果这个程序已经存入我们计算机的次级存储器（磁盘），通过双击 Word 程序图标等方式可以启动这个程序， 导致该程序从磁盘被加载到主存储器中。然后 CPU 逐条取出该程序的指令并执行，直至最后 一条指令执行完毕，程序即告结束。在执行过程中，有些指令会导致与用户的交互，例如用 户利用键盘输入或删除文字，利用鼠标点击菜单进行存盘或打印等等。就这样，通过执行成 千上万条简单的指令，最终解决了用计算机写文章的问题。

针对一个问题，设计出解决问题的程序（指令序列），并由计算机来执行这个程序，这 就是计算（computation）。

通过计算，使得只会执行简单操作的计算机能够完成神奇的复杂任务，所以计算机的神 奇表现其实都是计算的威力。如果读者对计算的能力还有疑问，下面这个例子或许能打消这 个疑问。Lucy 是一个只学过加法的一年级小学生，她能完成一个乘法运算任务吗？答案是肯 定的！解决问题的关键在于编写出合适的指令序列让 Lucy 机械地执行。例如下列“程序” 就能使 Lucy 算出 m×n：

```
在纸上写下 0，记住结果；
给所记结果加上第 1 个 n，记住结果；
给所记结果加上第 2 个 n，记住结果；
……
给所记结果加上第 m 个 n，记住结果。至此就得到了 m×n。
```

不难看出，这个指令序列的每一步都是 Lucy 能够做到的，因此最后确实能完成乘法运 算。这就是“计算”所带来的成果①。

计算机就是通过这样的“计算”来解决所有复杂问题的。执行大量简单指令组成的程序 虽然枯燥繁琐，但计算机作为一种机器，其特长正在于机械地、忠实地、不厌其烦地执行大 量简单指令！

> ① 当然，这种计算看上去就很繁琐，原因在于用到的基本指令（加法）太简单。如果 Lucy 学习了更“高级” 的指令（如乘法口诀等），就可以更快捷地完成乘法运算。

计算机的通用性

通过前面的介绍，可知计算机就是进行“计算”的机器。显然，这里的“计算”并不是日常说的数学计算。事实上，计算机在屏幕上显示信息，在 Word 文档中查找并替换文本， 播放 mp3 音乐，这些都是计算。

理解了计算机是如何计算的，也就能理解为什么计算机具有通用性，能解决各种不同类 型的问题。其中的奥秘就在于程序。如果想用计算机写文章，就将 Word 之类的程序加载到 主存中让 CPU 去执行，这时计算机就成了一台电子打字机；如果想用计算机听音乐，就将 Media Player 之类的程序加载到主存中让 CPU 去执行，这时计算机就成了一台音频播放机； 如果将 IE 之类的程序加载到主存中让 CPU 去执行，计算机就可以在互联网上浏览信息。总 之，一台计算机的硬件虽然固定不变，但通过加载执行不同的程序，就能实现不同的功能， 解决不同的问题。

我们平时说的计算机都是指通用计算机，能够安装执行各种不同的程序。其实在工业控 制和嵌入式设备等领域，也存在专用计算机，它们只执行预定的程序，从而实现固定的功能。 例如号称电脑控制的洗衣机，其实就是能执行预定程序的计算机。

计算机科学

为了更好地利用计算机解决问题，人们深入研究了关于计算的理论、方法和技术，形成 了专门研究计算的学问——计算机科学（computer science）①。

计算机科学包含很多内容，本书的主题是计算机科学家在用计算机解决问题时建立的一 些思想和方法，这些思想和方法普遍存在于计算机科学的各个分支之中。作为例子，我们来 看计算机科学家思考的一个根本问题：到底什么问题是计算机可计算的？一般人会以为，一 个问题能不能用计算机计算，取决于该计算机的计算能力；而计算机的计算能力又取决于 CPU 的运算速度、指令集、主存储器容量等硬件指标。真如此的话，显然巨型计算机应该具 有比微型计算机更强大的计算能力。然而，作为计算机科学理论基础的可计算性理论却揭示 了一个出人意料的结论：所有计算机的计算能力都是一样的！尽管不同计算机有不同的指令 集和不同性能的硬件，但一台计算机能解决的问题，另一台计算机肯定也能解决。

1.1.2 计算机语言

如前所述，计算机解决问题的过程实质上是机械地执行人们为它编制的指令序列的过程。为了告诉计算机应当执行什么指令，需要使用某种计算机语言。这种计算机语言能够精 确地描述计算过程，称为程序设计语言或编程语言（programming language）。

与计算机打交道的理想语言当然是像科幻电影所展示的那样，人类用自然语言与计算机（电影中更多的是机器人）进行对话。遗憾的是，由于自然语言的词语和句子往往有歧义， 既不精确也不简练，至少目前的计算机还不能很好地理解自然语言。所以计算机科学家设计 了人造语言来与计算机进行交流。编程语言是人工设计的形式语言，具有严格的语法和语义， 因此没有歧义的问题。

机器语言

CPU 制造商在设计某种 CPU 硬件结构的同时，也为其设计了一种“母语”——指令集， 这种语言称为机器语言（machine language）。机器语言在形式上是二进制的，即所有指令都 是由 0 和 1 组成的二进制序列。利用机器语言写的程序自然就是二进制指令的序列。我们来

> ① 不能望文生义地以为计算机科学是关于计算机的学问。著名计算机科学家 Dijkstra 有一句名言：计算机之 于计算机科学，正如望远镜之于天文学。

看一条 Intel 8086 处理器的机器指令：

```
0000010000000001
```

只要你将这一串 0/1 序列交给 CPU，CPU 就会按指令要求执行特定操作——将 1 存储 到计算机的某个寄存器①当中。计算机只懂得这种非常低级的机器语言。显然，用机器语言 编程序与计算机打交道，实在是太麻烦了，毕竟机器语言指令既难理解又难记忆。

汇编语言

为了使编程更容易，人们发明了汇编语言（assembly language）。汇编语言本质上是将机 器指令用更加容易为人们所理解和记忆的“助忆符”形式表现出来。例如前面那条将 1 存入 寄存器的机器指令在汇编语言中可以写成：

```
MOV AL, 1
```

可见在汇编语言中，指令的操作符是用 MOV（即 move）之类的助忆符表示的，操作数 据也用易理解的数字或符号来表示，因此指令的含义变得非常容易理解，例如上面这条指令 可以读成“将 1 送入寄存器 AL”。虽然编写汇编语言程序对程序员来说难度降低了很多，但 是很遗憾，计算机并不懂汇编语言。为了使计算机理解汇编语言程序，需要用一种称为汇编 器（assembler）的程序把汇编语言程序翻译成机器语言程序。有了汇编器这个“翻译”，程 序员“说”的汇编语言就能被计算机“听”懂并执行了。

即使到了今天，汇编语言在某些场合（如嵌入式系统）仍然非常有用，因为用汇编语言 能够写出执行效率很高的程序。但是，汇编语言和机器语言并没有本质上的差别，同样属于 非常低级的语言。而低级语言具有无法克服的缺点：第一，低级语言与机器硬件结构紧密关 联，因此为掌握低级语言必须了解很多底层硬件知识，导致低级语言的学习和使用都很困难， 开发效率低而且容易出错；第二，由于不同硬件的计算机具有不同的机器语言和汇编语言， 一类计算机上的低级语言程序不能拿到另一类计算机上执行，我们说低级语言程序不具有可 移植性。

高级编程语言

为了克服低级语言的缺点，计算机科学家设计出了更加易用的高级编程语言（high-level programming language）。高级语言相对于机器语言和汇编语言具有很多优点：第一，高级语 言吸收了人们熟悉的自然语言（英语）和数学语言的某些成分，因此非常易学、易用、易读； 第二，高级语言在构造形式和意义方面具有严格定义，从而避免了语言的歧义性；第三，高 级语言与计算机硬件没有关系，用高级语言写的程序可以移植到各种计算机上执行。

如果用高级语言来表达将 1 存入某处的指令，可以写成这样：

```
x = 1
```

显然这更加类似于我们从小就熟悉的数学语言，很容易理解和学会使用。

编译和解释

用高级语言所写的程序是不能直接交给计算机执行的，因为计算机完全不懂 x = 1 之 类的语句。为了让计算机理解并执行，必须先将高级语言程序翻译成机器语言程序。

高级语言的翻译有两种方式：编译和解释。

编译器（compiler）将高级语言程序（称为源代码）完整地翻译成等价的机器语言程序（称为目标代码），如图 1.2 所示。编译的特点是“一劳永逸”，整个源代码一旦翻译完毕， 今后就可以在任何时候多次执行目标代码，再也不需要编译器的参与了。就像翻译家将一本 英文小说笔译成中文，这是一次性的工作，作为翻译结果的中译本可以多次阅读。以编译方

> ① 寄存器是 CPU 里面的高速存储部件。

式处理源代码，对目标代码可以进行很多细致的优化，从而程序的执行速度一般会更快。就像翻译家对中译本可以精雕细琢，从而达到信达雅的境界。

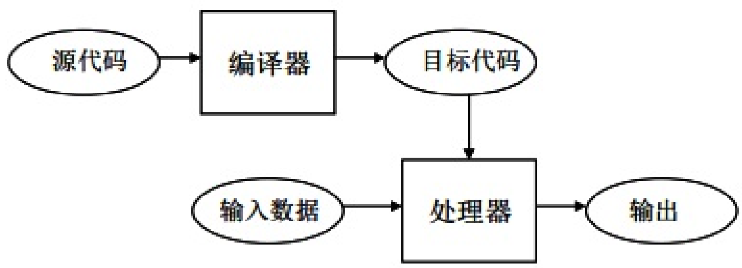


图 1.2 高级语言的编译

解释器（interpreter）直接分析并执行高级语言程序，如图 1.3 所示。解释的特点是“见 招拆招”，对源代码总是临机进行解释和执行。就像外交部的口译人员所做的工作，国家主席 说一句中文，口译者立即将它翻译成英文；即使主席后来说了同样的话，口译者还是要重新 翻译，无法重复利用以前的翻译结果。解释执行的处理方式无法进行上下文信息来进行优化，导致程序执行速度较慢，正如口译者无法琢磨最佳译文一样。但解释性语言具有更灵活的编程环境，可以交互式地输入程序语句并立即执行，程序员面对的仿佛是一台能听懂高级语言的计算机。

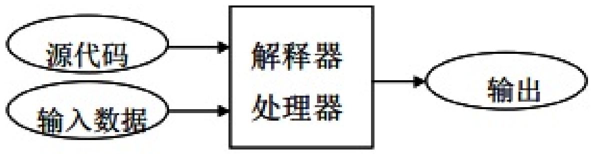


图 1.3 高级语言的解释

高级语言之所以具有前面提到的可移植性，正是因为高级语言的这种先翻译后执行的特 点。只要一台计算机上有合适的编译器或解释器，用某种高级语言编写的程序可以在该计算 机上执行。就像国家主席的讲话可以被中译英口译人员翻译给英语国家的人听，也可以被中 译法口译人员翻译给法语国家的人听一样。

还要说明的是，编译器和解释器本身也是程序，这种程序所执行的计算就是将别的程序 翻译成机器能够理解的指令。为了让一台计算机能够执行某种高级语言程序，必须先在该计 算机上安装特定高级语言的编译器或解释器程序！

迄今为止，计算机科学家们发明了数百种高级编程语言。不同语言的细节不尽相同，但 一些基本语言构造在绝大多数语言中都是存在的，例如输入输出、基本的数学运算、有条件 地执行和重复地执行等等。一般只要掌握一种编程语言，就足以利用计算机去解决实际问题。 而且一旦掌握了一种编程语言，再去学习其他语言也会变得非常容易。

本书要讨论的是用计算机解决问题时的思想和方法，这些内容原则上与使用哪种编程语 言没有关系。但是，为了更好地掌握本书的内容，需要进行编程实践，这就要求我们必须学 会某种编程语言。选择什么编程语言呢？高级编程语言虽多，但流行的并没有多少。2012 年 4 月公布的 TIOBE 编程语言排行榜①上，位列前 10 名的语言分别是 C、Java、C++、Objective-C、 C#、PHP、（Visual）Basic、Python、JavaScript 和 Perl。本书将采用位列其中的 Python 语言， 选择这个语言的理由是该语言非常易学易用，而且特别适合教学。

> ① http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

1.1.3 算法

如前所述，程序是解决某个问题的指令序列。编程解决一个问题时，首先要找出解决问 题的方法，该解决方法一般先以非形式化的方式表述为由一系列可行的步骤组成的过程，然 后才用形式化的编程语言去实现该过程。这种解决特定问题的、由一系列明确而可行的步骤 组成的过程，称为算法（algorithm①）。算法表达了解决问题的核心步骤，反映的是程序的解 题逻辑。

算法其实并不是随着计算机的发明才出现的东西。例如，早在两千多年前，古希腊数学 家欧几里德就发明了一种求两个自然数的最大公约数的过程，这个过程被认为是史上第一个 算法②：

【欧几里德算法】

```
输入：自然数 a、b
输出：a、b 的最大公约数
步骤：
    第 1 步：令 r 为 a / b 所得余数
    第 2 步：若 r = 0，则算法结束，b 即为答案；否则置 a←b，b←r，转到第 1 步。
```

又如，我们在小学学习的竖式乘法、长除法等等其实也都是算法的例子，都是通过明确 定义的一步一步的过程来解决问题。

利用计算机解决问题的关键就在于设计出合适的算法，当今计算机在各行各业中的成功 应用从根本上说都取决于高效算法的发现。例如，数学家发明了“充电放电”算法，从而利 用计算机证明了著名的四色定理。又如，谷歌公司的创建者发明了更合理的网页相关性排名 算法，从而使 Google 成为最成功的搜索引擎。其他如 MP3 播放器等便携式电子产品依靠聪 明的音频视频压缩算法来节省存储空间，GPS 导航仪利用高效的最短路径算法来规划最短路 线等等，不一而足。

算法是由一系列步骤构成的计算过程，但并不是随便用一些步骤都能构成合格的算法 的。我们对算法有两个要求：第一，每个步骤必须具备明确的可操作性；第二，构成算法的 所有步骤必须能在有限时间内完成。

先看算法步骤的可操作性。从最底层来看，计算机指令集中的基本指令显然具有可操作 性，因为 CPU 确定无疑地能够执行这些指令。然而，由于用简单的机器指令来表达算法步骤 会使得算法琐碎冗长，不能凸显算法表达的解题逻辑，所以实际上我们会用更高级别的操作 来表达算法步骤。打个比方，如果在菜谱中使用非常细节化的指令，那么在很多菜的菜谱中 都会看到这样的步骤：

```
......
冷水入锅
点火将水烧开
某蔬菜入锅
等水再次烧开
捞出蔬菜备用
......
```

这种步骤虽然详细，但太琐碎了。几乎没有菜谱会在这样的细节级别上表达操作步骤，一般都会将这个过程写做：

```
......
将某蔬菜绰水
......
```

> ① 这个词源自 9 世纪波斯数学家 Muhammad ibn Musa al-Khwarizmi 的姓（拉丁语写法 Algorismus）。
>
> ② 中国称为辗转相除法。

其中“绰水”是一个较高级别的指令，它本身又由若干更细节化的步骤组成，但它显然是一 个明确定义的步骤，按照这样的菜谱去做菜，完全是可操作的。

再以数学为例，“计算 b2 - 4ac”作为算法中的一个步骤显然是可操作的，没有必要细化 为“先计算 b2，再计算 4ac，再两者相减”的步骤；而“作一条平行于直线 AB 的直线”就 不是一个明确的步骤。至于“用尺规作图来三等分角ÐAOB”，则根本就是一个不可能做到的 操作，尽管其意义是明确的。

总之，在设计算法时，要选择合适的细节级别的步骤，不但要确保所有步骤处于计算机 能力范围之内，还应该使算法的读者容易理解算法的逻辑。

再看算法的有限性。只满足算法步骤的可操作性是不够的，一个合格的算法还必须能在 有限时间内执行完毕。例如，具备一点数论知识的人都知道“检查自然数 n 是不是质数”是 可行的步骤，例如可以逐个检查从 2 到 n/2 的自然数是不是 n 的因子。那我们能不能设计如 下“算法”来生成所有质数呢？

```
第 1 步：令 n = 2
第 2 步：检查 n 是不是质数
第 3 步：如果是就输出 n
第 4 步：n = n + 1，转到第 2 步
```

很遗憾，这不是一个合格的算法，因为自然数有无穷多个，导致“算法”的第 4 步是可

以无限进行下去的。 那么对一个给定的问题，能不能找到符合上述要求的算法呢？这其实正是计算机科学要

回答的一个基本问题：什么是可计算的？如果能够为某个问题找到算法，该问题就称为可计算的。当然，如果没能找到算法，并不意味着该问题不可计算，那也许只是因为我们不够聪明而已。事实上，计算机科学还从理论上对可计算性和计算复杂性进行分析。本书第 x 章会告诉大家，有一些看似简单的问题实际上不存在算法，而另一些问题虽然有算法但需要天文数字的时间和空间来完成计算，从而毫无实际价值。

1.1.4 实现

给定一个问题，当我们找到解决问题的算法后，接着就需要用某种计算机语言将这个算法表达出来，最终得到一个能被计算机执行的程序（或代码），这个过程称为实现（implementation），或者俗称为写代码（coding）。严格地说，算法与程序是不同的：算法是用非形式化方式表述的解决问题的过程，程序

则是用形式化编程语言表述的精确代码。这样，算法设计和算法实现就分别指计算机解决问题时的两个不同阶段。但我们经常在宽泛的意义上使用“程序”和“程序设计”这两个术语，前者泛指算法和代码，后者泛指从问题分析直到编码实现的全过程。

设计算法是创造性的活动，要求设计者具备问题求解能力和想像力，能从宏观视野把握问题的求解逻辑；而编码实现算法则是相对机械的活动，要求程序员具有严谨细致的作风，能在微观层次关注细枝末节。

可见，程序设计这项活动具有一定的挑战性，但这并不意味着只有非常聪明的人才能学习掌握程序设计。事实上，通过理论学习和动手实践，从小学生到大学生都能学会程序设计。

学习程序设计有很多好处。第一，计算机已经成为我们生活、学习和工作中普遍使用的工具，学会编程能使我们成为计算机的主人，使计算机按我们的意志做事。第二，程序设计能够培养我们抽象、分析和问题求解的能力，这种能力对日常生活和工作也是很重要的。第三，编程也是一种充满乐趣的智力活动，许多人将编程当作一种爱好，发现巧妙的算法和程序运行成功后的那种成就感令人乐此不疲。

1.2 什么是计算思维？

如前所述，计算是利用计算机一步一步地执行指令来解决问题的过程，计算机科学是关于计算的科学。正如数学家在证明数学定理时有独特的数学思维、工程师在设计制造产品时 有独特的工程思维、艺术家在创作诗歌音乐绘画时有独特的艺术思维一样，计算机科学家在 用计算机解决问题时也有自己独特的思维方式和解决方法，我们统称之为 计算思维（computational thinking）。从问题的计算机表示、算法设计直到编程实现，计算思维贯穿于 计算的全过程。学习计算思维，就是学会像计算机科学家一样思考和解决问题。

1.2.1 计算思维的基本原则

计算思维建立在计算机的能力和限制之上，这是计算思维区别于其他思维方式的一个重

要特征。用计算机解决问题时必须遵循的基本思考原则是：既要充分利用计算机的计算和存储能力，又不能超出计算机的能力范围。

例如，能够高速执行大量指令是计算机的能力，但每条指令只能进行有限的一些简单操 作则是计算机的限制，因此我们不能要求计算机去执行无法化归为简单操作的复杂任务。又 如，计算机只能表示固定范围的有限整数，任何算法如果涉及超出范围的整数，都必须想办 法绕开这个限制。再如，计算机的主存速度快、容量小、靠电力维持存储，而磁盘容量大、 不需要电力维持存储但存取速度慢，因此涉及磁盘数据的应用程序必须寻求高效的索引和缓 冲方法来处理数据，以避免频繁读写磁盘。

虽然计算思维有自己的独特性，但它同时也吸收了其他领域的一些思维方式。例如，计 算机科学家像数学家一样建立现实世界的抽象模型，使用形式语言表达思想；像工程师一样 设计、制造、组装与现实世界打交道的产品，寻求更好的工艺流程来提高产品质量；像自然 科学家一样观察系统行为，形成理论，并通过预测系统行为来检验理论；像经济学家一样评 估代价与收益，权衡多种选择的利弊；像手工艺人一样追求作品的简洁、精致、美观，并在 作品中打上体现本人风格的烙印。

计算思维是人的思想和方法，旨在利用计算机解决问题，而不是使人类像计算机一样做 事。作为“思想和方法”，计算思维是一种解题能力，一般不是可以机械地套用的，只能通过 学习和实践来培养。计算机虽然机械而笨拙，但人类的思想赋予计算机以活力，装备了计算 机的人类利用自己的计算思维能够解决过去无法解决的问题、建造过去无法建造的系统。

1.2.2 计算思维的具体例子

基于计算机的能力和局限，计算机科学家提出了很多关于计算的思想和方法，从而建立

了利用计算机解决问题的一整套思维工具。下面我们简要介绍计算机科学家在计算的不同阶段所采用的常见思想和方法。

问题表示

用计算机解决问题，首先要建立问题的计算机表示。问题表示与问题求解是紧密相关的， 如果问题的表示合适，那么问题的解法就可能如水到渠成一般容易得到，否则可能如逆水行 舟一般难以得到解法。

抽象（abstraction）是用于问题表示的重要思维工具。例如，小学生经过学习都知道将 应用题“原来有五个苹果，吃掉两个后还剩几个”抽象表示成“5 - 2”，这里显然只抽取了 问题中的数量特性，完全忽略了苹果的颜色或吃法等不相关特性。一般意义上的抽象，就是 指这种忽略研究对象的具体的或无关的特性，而抽取其一般的或相关的特性。计算机科学中 的抽象包括数据抽象和控制抽象，简言之就是将现实世界中的各种数量关系、空间关系、逻

辑关系和处理过程等表示成计算机世界中的数据结构（数值、字符串、列表、堆栈、树等）和控制结构（基本指令、顺序执行、分支、循环、模块等），或者说建立实际问题的计算模型。 另外，抽象还用于在不改变意义的前提下隐去或减少过多的具体细节，以便每次只关注少数 几个特性，从而有利于理解和处理复杂系统。显然，通过抽象还能发现一些看似不同的问题 的共性，从而建立相同的计算模型。总之，抽象是计算机科学中广泛使用的思维方式，只要 有可能并且合适，程序员就应当使用抽象。

可以在不同层次上对数据和控制进行抽象，不同抽象级对问题进行不同颗粒度或详细程 度的描述。我们经常在较低抽象级之上再建立一个较高的抽象级，以便隐藏低抽象级的复杂 细节，提供更简单的求解方法。例如，对计算本身的理解就可以形成“电子电路®门逻辑® 二进制®机器语言指令®高级语言程序”这样一个由低到高的抽象层次，我们之所以在高级 语言程序这个层次上学习计算，当然是为了隐藏那些低抽象级的繁琐细节。又如，在互联网 上发送一封电子邮件实际上要经过不同抽象级的多层网络协议才得以实现，写邮件的人肯定 不希望先掌握网络低层知识才能发送邮件。再如，我们经常在现有软件系统之上搭建新的软 件层，目的是隐藏低层系统的观点或功能，提供更便于理解或使用的新观点或新功能。

算法设计

问题得到表示之后，接下来的关键是找到问题的解法——算法。算法设计是计算思维大 显身手的领域，计算机科学家采用多种思维方式和方法来发现有效的算法。例如，利用分治 法的思想找到了高效的排序算法，利用递归思想轻松地解决了 Hanoi 塔问题，利用贪心法寻 求复杂路网中的最短路径，利用动态规划方法构造决策树，等等。前面说过，计算机在各个 领域中的成功应用，都有赖于高效算法的发现。而为了找到高效算法，又依赖于各种算法设 计方法的巧妙运用。

对于大型问题和复杂系统，很难得到直接的解法，这时计算机科学家会设法将原问题重 新表述，降低问题难度，常用的方法包括分解、化简、转换、嵌入、模拟等。如果一个问题 过于复杂难以得到精确解法，或者根本就不存在精确解法，计算机科学家不介意退而求其次， 寻求能得到近似解的解法，通过牺牲精确性来换取有效性和可行性，尽管这样做的结果可能 导致问题解是不完全的，或者结果中混有错误。例如搜索引擎，它们一方面不可能搜出与用 户搜索关键词相关的所有网页，另一方面还可能搜出与用户搜索关键词不相关的网页。作为 对比，很难想象数学家在解决数学问题时会寻求什么近似证明或对错参杂的解。

编程技术

找到了解决问题的算法，接下来就要用编程语言来实现算法，这个领域同样是各种思想 和方法的宝库。例如类型化与类型检查方法将待处理的数据划分为不同的数据类型，编译器 或解释器借此可以发现很多编程错误，这和自然科学中的量纲分析的思想是一致的。又如结 构化编程方法使用规范的控制流程来组织程序的处理步骤，形成层次清晰、边界分明的结构 化构造，每个构造具有单一的入口和出口，从而使程序易于理解、排错、维护和验证正确性。 又如模块化编程方法采取从全局到局部的自顶向下设计方法，将复杂程序分解成许多较小的 模块，解决了所有底层模块后，将模块组装起来即构成最终程序。又如面向对象编程方法以 数据和操作融为一体的对象为基本单位来描述复杂系统，通过对象之间的相互协作和交互实 现系统的功能。还有，程序设计不能只关注程序的正确性和执行效率，还要考虑良好的编码 风格（包括变量命名、注释、代码缩进等提高程序易读性的要素）和程序美学问题。

编程范型（programming paradigm）是指计算机编程的总体风格，不同范型对编程要素（如数据、语句、函数等）有不同的概念，计算的流程控制也是不同的。早期的命令式（或 称过程式）语言催生了过程式（procedural）范型，即一步一步地描述解决问题的过程。后来

发明了面向对象语言，数据和操作数据的方法融为一体（对象），对象间进行交互而实现系统 功能，这就形成了面向对象（object-oriented）范型。逻辑式语言、函数式语言的发明催生了 声明式（declarative）范型——只告诉计算机“做什么”，而不告诉计算机“怎么做”。有的语言只支持一种特定范型，有的语言则支持多种范型。本书采用的 Python 就是支持多种编程范型的语言，Python 程序可以是纯过程式的，也可以是面向对象的，甚至可以是函数式的。

可计算性与算法复杂性

在用计算机解决问题时，不仅要找出正确的解法，还要考虑解法的复杂度。这和数学思维不同，因为数学家可以满足于找到正确的解法，决不会因为该解法过于复杂而抛弃不用。 但对计算机来说，如果一个解法太复杂，导致计算机要耗费几年几十年乃至更久才能算出结 果，那么这种“解法”只能抛弃，问题等于没有解决。有时即使一个问题已经有了可行的算 法，计算机科学家仍然会去寻求更有效的算法。

有些问题是可解的但算法复杂度太高，而另一些问题则根本不可解，不存在任何算法过 程。计算机科学的根本任务可以说是从本质上研究问题的可计算性。例如，科幻电影里的计 算机似乎都像人类一样拥有智能，从计算的本质来说，这意味着人类智能能够用算法过程描 述出

来。虽然现代计算机已经能够从事定理证明、自主学习、自动推理等“智能”活动，但 是人类做这些事情并非采用一步一步的算法过程，像阿基米德大叫“尤里卡”那样的智能活 动至少目前的计算机是没有可能做到的。

虽然很多问题对于计算机来说难度太高甚至是不可能的任务，但计算思维具有灵活、变 通、实用的特点，对这样的问题可以去寻求不那么严格但现实可行的实用解法。例如，计算 机所做的一切都是由确定性的程序决定的，以同样的输入执行程序必然得到同样的结果，因 此不可能实现真正的“随机性”。但这并不妨碍我们利用确定性的“伪随机数”生成函数来模 拟现实世界的不确定性、随机性。

又如，当计算机有限的内存无法容纳复杂问题中的海量数据时，这个问题是否就不可解 了呢？当然不是，计算机科学家设计出了缓冲方法来分批处理数据。当许多用户共享并竞争 某些系统资源时，计算机科学家又利用同步、并发控制等技术来避免竞态和僵局。

1.2.3 日常生活中的计算思维

人们在日常生活中的很多做法其实都和计算思维不谋而合，也可以说计算思维从生活中吸收了很多有用的思想和方法。我们来看一些例子。 算法过程：菜谱可以说是算法（或程序）的典型代表，它将一道菜的烹饪方法一步一步地罗列出来，即使不是专业厨师，照着菜谱的步骤也能做出可口的菜肴。这里，菜谱的每一 步骤必须足够简单、可行。例如："将土豆切成块状"、"将 1 两油入锅加热"等都是可行的步 骤，而"使菜肴具有神秘香味"则不是可行的。

模块化：很多菜谱都有"勾芡"这个步骤，与其说这是一个基本步骤，不如说是一个模 块，因为勾芡本身代表着一个操作序列——取一些淀粉，加点水，搅拌均匀，在适当时候倒 入菜中。由于这个操作序列经常使用，为了避免重复，也为了使菜谱结构清晰、易读，所以 用"勾芡"这个术语简明地表示。这个例子同时也反映了在不同层次上进行抽象的思想。

查找：如果要在英汉词典中查一个英文单词，相信读者不会从第一页开始一页页地翻看， 而是会根据字典是有序排列的事实，快速地定位单词词条。又如，如果现在老师说请将本书 翻到第 8 章，读者会怎么做呢？是的，书前的目录可以帮助我们直接找到第 8 章所在的页码。这正是计算机中广泛使用的索引技术。

回溯：人们在路上遗失了东西之后，会沿原路边往回走边寻找。或者在一个岔路口，人 们会选择一条路走下去，如果最后发现此路不通就会原路返回，到岔路口选择另一条路。这种回溯法对于系统地搜索问题空间是非常重要的。 缓冲：假如将学生用的教科书视为数据，上课视为对数据的处理，那么学生的书包就可以视为缓冲存储。学生随身携带所有的教科书是不可能的，因此每天只能把当天要用的教科 书放入书包，第二天再换入新的教科书。

并发。厨师在烧菜时，如果一个菜需要在锅中煮一段时间，厨师一定会利用这段时间去 做点别的事情（比如将另一个菜洗净切好），而绝不会无所事事。在此期间如果锅里的菜需要 加盐加佐料，厨师可以放下手头的活儿去处理锅里的菜。就这样，虽然只有一个厨师，但他 可以同时做几个菜。

类似的例子还有很多，在此就不一一列举了。要强调的一点是，读者在学习用计算机解 决问题的时候，如果经常想想生活中遇到类似问题时的做法，一定会对找出问题解法有所帮 助。

1.2.4 计算思维对其他学科的影响

随着计算机在各行各业中得到广泛应用，计算思维对许多学科都产生了重要影响。下面以数学、生物学和化学为例进行简单的介绍。 数学：计算机对数学来说过去只是一个数值计算工具，用于快速、大规模的数值计算，

对数值计算方法的研究导致了计算数学的形成。后来数学家利用计算机进行代数演算，形成了计算机代数；利用计算机研究几何问题，形成了计算几何学。数学家还利用计算机去验证数学猜想，虽然不能证明猜想，但是一旦发现反例就可以推翻猜想，以免数学家毕生投入到一个不成立的猜想之中。在定理证明方面，美国数学家通过设计算法过程来验证构型，最终证明了著名的四色定理；我国的吴文俊院士更是建立了初等几何和微分几何定理的机械化证明方法，为数学机械化开辟了方向。总之，现在计算机已经成为数学的研究手段，大大扩展了数学家的能力。

生物学：计算机和万维网迅速而显著地改变了生物学研究的面貌，过去生物学家在实验 室进行的研究现在可以在计算机上进行，因此出现了生物信息学（较老的叫法是计算生物学） 这一学科。生物信息学的内容包括基因组测序、建立基因数据库、发现和查找基因序列模式 等等，这一切都有赖于计算技术的应用。生物信息学的发展正在改变着生物学家的思维方式，他们除了研究生物学，还研究高效的算法。对生物信息学家来说，对生物学的理解和对计算的理解同等重要。

化学：计算技术对公认的纯实验科学——化学也产生了巨大影响，化学的研究内容、研 究方法甚至学科的结构和性质都发生了深刻变化，从而形成了计算化学这一交叉学科。计算 化学的主要研究内容包括分子结构建模与图像显示、计算机分子模拟、计算量子化学、分子CAD、化学数据库等，能够帮助化学家在原子分子水平上阐明化学问题的本质，在创造特殊性能的新材料、新物质方面发挥重大的作用。

此外，计算物理学、计算博弈论、计算材料学、计算广告学、电子商务等等新学科也都 在蓬勃发展。可以预见，“计算+X”将成为很多学科的发展方向之一。

1.3 初识 Python

1.3.1 Python 简介

Python 是一种通用的高级编程语言，由荷兰人 Guido van Rossum 于 1980 年代发明①。

前面说过，高级编程语言有数百种，而 Python 跻身流行语言的前 10 名之中。与其他语言相比，Python 的主要特点包括：

> ① Python 这个名字源自发明者喜欢的电视喜剧节目 Monty Python's Flying Circus，而不是什么爬行动物。

- Python 语言最重要的设计理念是追求高度的可读性。与大多数语言不同，Python 语言的语法要求程序代码具有整齐而有条理的形式，代码的外在形式与内在意义紧密 相关。这样做的好处是：外观不整齐的代码属于编程错误，从而提醒编程人员避免 很多错误。
- Python 语言的另一个设计理念是尽量避免“这件事可以有多种做法”，因此语言中 冗余的成分很少，程序员经常只有唯一的也是最好的语言构造可用。
- Python 语言同时支持过程式、面向对象式和函数式等多种编程范型，拥有丰富的标 准库来支持应用开发所需的各种功能。

这些设计理念导致 Python 语法简明易学，代码清晰美观、易读易理解。Python 语言的众 多优点使得它在编程者中越来越流行，并使它在 2007 年和 2010 年两次获得 TIOBE 年度编程语言奖。

Python 是解释型语言，Python 语句或程序（.py 文件）首先被解释器翻译成字节码（byte code），然后再由 Python 虚拟机来直接执行。

Python 的主要版本可分为 2.x 和 3.x 两类。Python 3.x 是最新的版本，代表 Python 的发 展方向，但问题是不兼容 2.x 版本。由于 2.7 版本包含了 3.x 版本的主要特征，所以本书选择 Windows 平台下的 Python 2.7 作为编程环境，本书所有例子都在此版本下测试通过，建议读者也下载安装这个版本①，以便在学习时能得到和本书中一样的结果。

读者花 1 分钟时间安装了 Python 2.7 之后，从“开始/所有程序/Python 2.7”中可以看到 有两种界面的解释器环境：命令行界面和图形用户界面（IDLE）。启动这两种界面之后所看 到的屏幕分别如图 1.4 和图 1.5 所示：

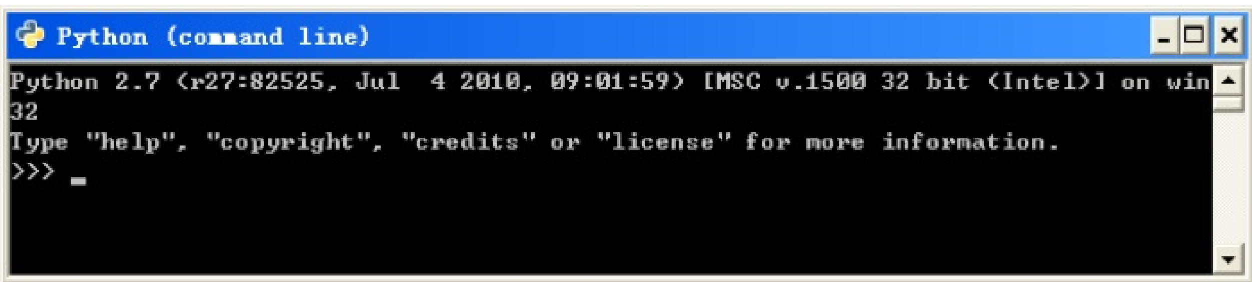


图 1.4 Python 命令行解释器环境

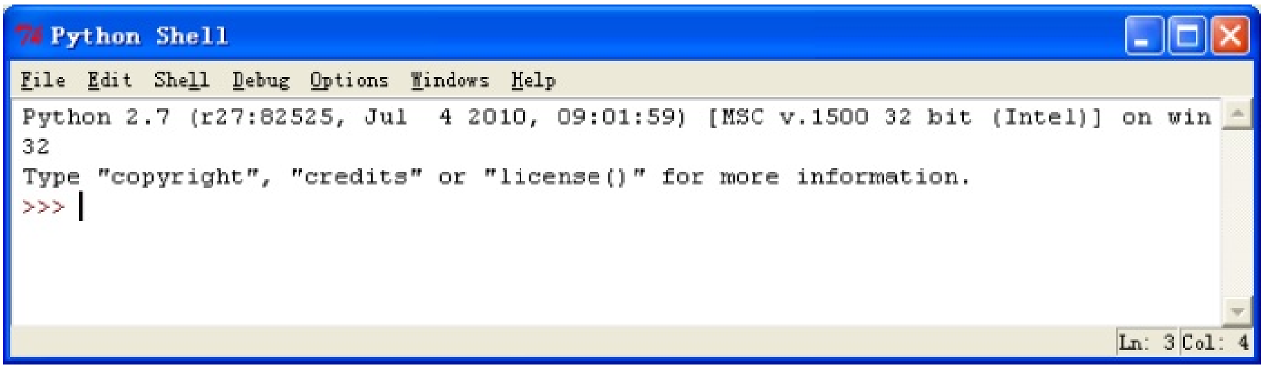


图 1.5 Python GUI 解释器环境

界面中的>>>是 Python 解释器的提示符，表示现在解释器已准备好执行程序。如果在提 示符后面输入 Python 语句，解释器将直接解释执行该语句。接下来我们就要开始学习 Python 语言的各种语句和各种编程方法，终极目标是让计算机按我们的指令做事。

> ① Python 官方网站：http://www.python.org/download/

1.3.2 第一个程序

学习一门编程语言时，传统上所写的第一个程序是 HelloWorld 程序，其功能是让计算机显示一句问候语"Hello, World!"。用 Python 来实现这个任务是非常简单的：首先启动 Python 解释器（命令行或 IDLE 均可，本书主要以 IDLE 界面为例），然后在提示符下输入下 面的内容

```
>>> print "Hello, World!"
Hello, World!
```

提示符>>>后面的黑体字部分是我们输入的 Python 语句，该语句的功能是在屏幕上显示信息， 在下一行我们看到了期望的输出。

曾有人根据 HelloWorld 程序的简单程度来判断一个编程语言的易学易用程度，按这个标 准，Python 可以说已经简单到了极致。

1.3.3 程序的执行方式

像上面 HelloWorld 程序所演示的那样，在 Python 解释器提示符>>>下输入语句并执行的 方式称为交互执行方式。

交互执行方式对执行单条语句来说是合适的，但是如果一个程序只有一条语句，那这个 程序肯定做不了什么大事。有用的程序都是由很多条语句组成的，而由很多条语句组成的程 序是不适合以交互方式执行的。例如，如果我们想让计算机在屏幕上连续显示三句问候语， 在交互方式下必然是这样的：

```
>>> print "Hello, Lucy."
Hello, Lucy.
>>> print "How are you?"
How are you?
>>> print "Goodbye, Lucy."
Goodbye, Lucy.
```

以上交互执行的结果明显不能令人满意，因为语句是每输入一条就执行一条，导致我们 希望连续显示的三句问候语被输入的程序语句分隔开了。

交互执行方式还有一个更严重的不足之处是：程序没有保存，语句一旦执行完就丢弃了， 因此无法多次执行一个程序。还是拿上面的例子来说，当用户想再次显示那三句问候语，他 就不得不重新输入所有语句。

为了解决上述问题，我们可以先将程序语句输入并保存在一个文件中，然后再“成批” 地执行程序文件中的所有语句。稍后的程序 1.2 给出了连续显示三句问候语的程序文件，执 行该程序文件即能看到连续显示的三句问候语，更重要的是该程序文件是“一次编写、永久 保存、多次执行”的。

Python 程序文件及其执行方式

将程序语句保存在一个扩展名为.py 的文本文件中，这种程序文件称为模块（module）。 还是以我们的第一个程序 HelloWorld 为例，先运行任何一种文本编辑器（如 Windows

的记事本①）新建一个文件，在文件中输入语句 print "Hello, World!" ，然后将文件 保存为 hello.py，这样就创建了一个 Python 程序文件（即模块）：

【程序 1.1】hello.py

```
print "Hello, World!"
```

接下来的问题是：如何执行模块文件 hello.py？在 Windows + Python 的平台上，我们有 多种可选择的方式。第一种方式是在 Windows 的命令提示符②下直接用 Python 解释器（即 python.exe 文件）执行该程序： C:\Python27> python hello.py Hello, World!

> ① 用 Word 之类的编辑器也可以，但要注意保存时必须选择纯文本格式。
>
> ② 俗称 DOS 界面。

效果如图 1.6（a）所示。

第二种方式是在 Python 解释器（命令行或 GUI）环境的提示符下执行 import 语句来“导入”程序文件，该语句的作用是将 Python 模块从磁盘加载到内存中，在加载的同时执行模块的每一条语句，就如在解释器环境下手工输入语句一样。具体语句形式如下：

```
>>> import hello
Hello, World!
```

效果如图 1.6（b）所示。注意，与第一种方式不同的是，模块文件名中不带扩展名“.py”， 因为 Python 自动假设模块具有“.py”扩展名。



（a）



（b）

图 1.6 Python 程序文件的执行方式

顺便说一下，第一次导入模块文件时，Python 会创建一个文件名相同但扩展名为.pyc 的 文件（本例中就是 hello.pyc），这是被 Python 解释器使用的一个中间文件——字节码文件。Python 语言的翻译采用的是编译、解释混合方式：模块文件中的 Python 源程序首先被编译成 较低级的字节码指令，然后再解释执行这些字节码指令。如果已经生成了.pyc 文件，以后再 导入相应的模块时速度就会更快，因为无需再次编译。反之，如果为了节省存储空间而删去.pyc 文件，下次导入模块时就需要重新编译。

第三种执行方式是直接在 Windows 文件夹中找到程序文件 hello.py，然后双击文件图标。 在时，Windows 系统首先打开一个 Python 解释器的命令行窗口（类似图 1.6（b）），然后执行 该程序，执行结束后 Windows 自动关闭命令行窗口。这种方式最简单直接，但它有一个让新 手困惑的小问题：由于程序执行的非常快，用户可能还没看清发生了什么，窗口就关闭了。 这个问题用一个小技巧就可以解决：在程序的最后放一条输入语句，该输入语句能让程序执 行完前面的语句后停顿下来等待用户输入，用户便有时间看清程序此前的运行结果。输入语 句的具体用法见第 2 章。

第四种方式是先在 IDLE 中打开（或者新建）程序文件 hello.py，具体打开方法可以利用 IDLE 菜单栏上的 File/Open...菜单项，也可以通过右键点击 hello.py 文件图标并选择“Edit with IDLE”菜单项。打开文件后进入 IDLE 自带的程序开发环境①窗口，在此窗口中选择 Run 菜 单中的 Run Module 命令（或直接按 F5 键）即可执行程序。注意程序的执行结果是显示在 Python 解释器提示符窗口中的。

> ① 程序开发环境是专为程序员使用某种语言编程而设计的系统，在其中可以方便地编辑、存储、执行、调试 程序。很多语言都有流行的集成开发环境（IDE）可用。IDLE 是标准的 Python 开发环境。

下面我们将前面提到的连续显示三条问候信息的程序保存到文件中：

【程序 1.2】eg1_2.py

```
print "Hello, Lucy."
print "How are you?"
print "Goodbye, Lucy."
```

如果以上述第四种方式来执行 eg1_2.py，会得到如图 1.7 所示的结果。从图中可见，显示的信息如我们所愿是连续的三句话。

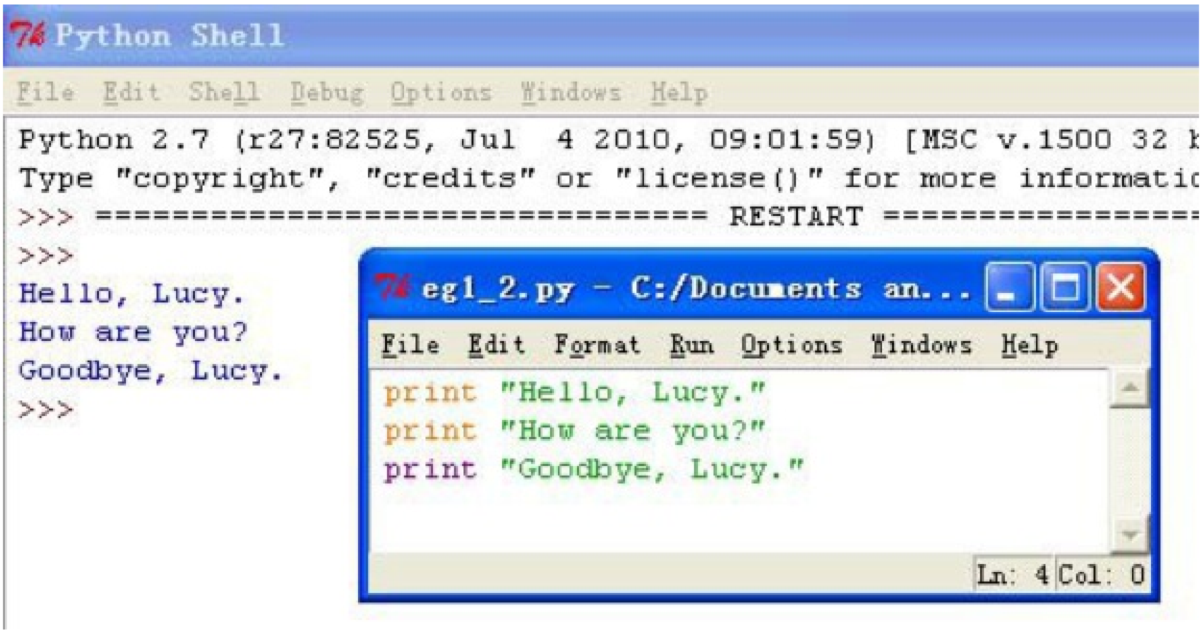


图 1.7 在 IDLE 中执行程序 1.2

在本书中，我们主要用两种方式来执行语句或程序。

当介绍 Python 语言的有关语句时，我们经常在 Python 解释器环境的提示符下以交互方 式逐条执行语句，因为这对特定内容的讲解和演示非常方便。作为标志，凡是用交互方式执 行的程序语句，我们都在语句前附上提示符“>>>”（注意：提示符不是程序的一部分！）。

当我们编写较大的完整程序时，一般都将程序保存为文件，然后再以前述任何方式来执 行该程序文件。为便于交叉引用，对于完整的程序文件，本书都会像程序 1.1 和程序 1.2 那样 专门编号，并给出程序文件名。

不管用交互方式还是批方式，强烈建议读者在阅读本书时亲自输入并执行这些语句或程 序。另外，建议读者使用 IDLE 的程序开发环境，因为 IDLE 是图形界面的环境，提供自动 缩进、用颜色区分语言成分等特性，能方便而高效地编辑、执行和调试程序。

Python 搜索路径 还有个问题是关于程序文件保存位置的。在前面提到的程序文件的第一、第二种执行方式下，我们实际上假设了

```
C:\Python27> python hello.py
```

和

```
>>> import hello
```

这两条命令都能找到要执行的程序文件 hello.py。事实上，如果我们将 hello.py 保存在 Python的安装目录 C:\Python27 之下，那这两条命令都能成功执行程序 hello.py。 但作为一种好的习惯，不应该将用户文件和系统文件混放在一起，因此我们通常都将自己的程序文件如 hello.py 保存在自己的目录中。这时如果还像上面两条命令这样通过文件名 来找程序文件，则要么 Windows 会报错说找不到指定文件，要么 Python 解释器会报错说找 不到指定模块。对此，一种解决方法是在文件名前面加上绝对路径，以告知系统该文件保存 在哪里，例如假设 D:\myPython 是我们保存程序文件的目录，则可以像下面这样执行程序：

```
C:\Python27> python D:\myPython\hello.py
```

或者

```
>>> import D:\myPython\hello
```

另一种更方便的做法是预先将 D:\myPython 添加到 Python 的搜索路径中，使系统仅根据 文件名就能找到程序文件。具体做法是，先用任何文本编辑器建立一个文本文件，其内容是 要添加的目录路径，例如：

```
D:\myPython
```

然后保存为扩展名为.pth 的文件，如“mypath.pth”，最后将这个文件复制到安装目录下的指 定子目录 C:\Python27\Lib\site-packages 中。

如果是以第三、第四种方式来执行程序文件，系统总是在当前工作目录中找文件，一般 不会出现找不到文件的问题。

1.3.4 Python 语言的基本成分

在自然语言中，我们用字词、句子、段落来写文章表达思想。类似地，编程语言也提供

各种语言成分用于构造程序表达计算。例如 HelloWorld 程序中的 print 是 Python 语言中用 于显示输出的一个保留词，而"Hello, World!"则是被显示的数据，这两个成分组合在一 起，就构成了一条完整的语句。本节简单介绍 Python 语言的基本成分，使读者对 Python 编 程有个概括的了解，更多细节将在本书后面的章节中介绍。

数据和表达式

程序是处理数据的，编程语言首先要有表达数据的语言成分，例如"Hello, World!" 就是被处理的数据。数据分为不同的类型，"Hello, World!"是字符串类型的数据。除了 字符串，Python 语言还能表达和处理数值型的数据，例如：

```
>>> print 3.14
3.14
```

Python 不但能表达"Hello,World!"和 3.14 这样的基本数据，还能表达数据运算。将 运算符施加到数据上所得到的语言构造称为表达式。例如下面的 print 语句显示一个表达 式的计算结果，该表达式中使用了加法（+）和乘法（*）运算符：

```
>>> print 2 + 3 * 4
14
```

变量与标识符

像"Hello, World!"和 3.14 这样的数据称为常量，其数据值由字面决定，并且不可 改变。Python 语言中还可以定义变量，用于表示可变的数据。变量具有名字，不同变量是通 过名字相互区分的，因此变量名具有标识作用，故称为标识符①。

Python 语言中，标识符的构成必须符合规则：以字母或下划线开头，后面跟随 0 个或多 个字母、数字、下划线。例如：

```
x xYz x1y2 xy_123 __ （连续两个下划线） _123
```

等都是合法的标识符，而

```
3q x-123 first name（中间用了空格）
```

等则是非法的。

> ① Python 程序中还有函数、类、模块等需要命名的构件，这些名字同样都属于标识符。

作为良好的编程风格，标识符的命名是有讲究的。首先，要尽量使用有意义的名字，例 如如果要用一个变量来表示工资，可以命名为 salary、gongzi 之类，而 s 或 gz 就不是好的名字。其次，如果用两个以上单词组成一个名字，最好能让人看出单词之间的分界，具 体做法有后续单词首字母大写①或者用下划线分隔等形式，例如表示出生年份的变量可以命 名为 birthYear 或 birth_year，而 birthyear 就不算是好的风格。第三，每个人应当 前后一致地使用某种命名风格，例如总是用后续单词首字母大写或总是用下划线分隔单词。 本书的示例程序中，一般以小写字母开头的一个或多个英文单词作为变量名，其中后续 单词的首字母都大写，例如 firstName、dateOfBirth。这也是很多人惯用的命名风格。 当然，在很多简单的示例程序中，我们也会使用很多无意义的单字母的变量名，毕竟这些程序不是正式的应用程序。

语句

语句是编程语言提供的基本命令，是程序的基本组成单元和执行单元。Python 语言提供

了多种语句，分别完成不同的功能，例如我们多次见到的 print 语句。每条语句都有规定 的语法形式和精确的语义，本书将采用“模板”的方式来介绍 Python 语句的语法。例如 print 语句的用法“模板”包括：

```
print <表达式>
print <表达式 1>, <表达式 2>, ..., <表达式 n>
```

在语句模板中我们用“<表达式>”之类的符号表示相应位置上所期待的合法语言成分。

第一个模板表示可以在 print 后面出现一个表达式，其含义是计算表达式的值并在屏幕上 显示计算结果。第二个模板表示 print 后面可以出现用逗号分隔的多个表达式，其含义是 计算每个表达式的值，并在屏幕的同一行上显示用空格分隔的各表达式的计算结果。例如：

```
>>> print "2 + 3 =", 2 + 3
2 + 3 = 5
```

最常用的一种语句是赋值语句，用于为变量赋值。最简单的赋值语句的形式是：

```
<变量> = <表达式>
```

其语义是先计算<表达式>的值，再将该值存储到<变量>中。例如：

```
>>> x = 2 + 3
```

执行结果是将 5 存储于变量 x 中，此后在表达式中使用 x 就相当于使用 5。例如：

```
>>> print x
5
>>> print x + 1
6
```

顾名思义，变量的值随时可以改变，例如下面的赋值语句将 x 的值从 5 改成了"Hello"：

```
>>> x = "Hello"
>>> print x
Hello
```

用 Python 语言编程时，通常是使每一条语句独占一行，而不将两条以上的语句写在同一 行上。如果一条语句很长，写在一行上读起来不方便，Python 也提供了“续行符”用于换行 继续输入：只要在一行的末尾输入字符“\”再按回车键，就表示本行语句未完，换到下一 行继续。例如：

```
>>> print "This is a very very looooooooooooooooooooooooooooong \
sentence."

This is a very very looooooooooooooooooooooooooooong sentence.
```

① 顺便提一下，首单词的首字母也大写习惯用于“类名”，而所有字母都大写习惯用于“常量名”。

函数

我们经常将一个语句序列定义成一个“函数”，从而将这个语句序列视为一个整体并命名。 今后在程序的任何地方，只要写下“函数名”，就相当于写下了构成该函数的语句序列，这称为“调用”该函数。例如，我们将程序 1.2 中的三条语句定义成一个函数：

```
>>> def greet():
        print "Hello, Lucy."
        print "How are you?"
        print "Goodbye, Lucy."
>>>
```

第一行的 def 告诉 Python 我们要定义一个函数，其后的 greet 是新定义的函数的名字，greet 后面的一对括号用于表示函数的参数。虽然本例中 greet 函数没有参数，但括号仍 然不可缺少。接下来三行是构成函数的语句序列，称为函数体。Python 语言要求：函数体中 的语句与 def 行相比，左边必须留一点空白（称为“缩进”），表示它们是函数的一部分。具 体缩进多少不重要，重要的是函数体的各语句左边要对齐。最后，交互方式下需要用一个空 行（在一行的开始处按回车键）来结束函数定义，使解释器回到提示符状态。

在此我们说明一下 Python 语言的缩进问题。一般来说，Python 程序中所有语句应该左对齐。但在某些情况下，下一行语句要比上一行语句左边多缩进一些空白，这是为了表达一种隶属关系：左缩进的语句是上面未缩进语句的下属部分。同层次的语句总是左对齐的，因此当下属部分结束后，后面的语句又要恢复到未缩进的状态。对于接触过其他编程语言的人来说，一开始也许会不习惯 Python 的代码缩进，但是以后会发现强制缩进的好处，例如程序在形式上更整齐、更容易排错等。

上面的函数定义只是告诉 Python 将来看到 greet 时应该做什么，现在并不执行函数体 中的语句序列。将来任何时候如果想执行函数的语句，只需输入函数名来“调用”函数，例 如：

```
>>> greet()
Hello, Lucy.
How are you?
Goodbye, Lucy.
```

注意函数名 greet 后面的一对括号，这是必须有的，表明这是一个函数调用。

作为惯例，一个 Python 程序中通常会定义一个名叫 main 的函数。对于简单程序，可以

将程序的所有语句放在 main 函数中；对于由很多函数组成的复杂程序，可以让 main 作为程序的执行入口。拿程序 1.2 来说，更常见的是以如下代码来编写：

```
def main():
    print "Hello, Lucy."
    print "How are you?"
    print "Goodbye, Lucy."
main()
```

注意最后一行的 main()，它的作用就是调用执行函数 main。没有这一行，该程序仅仅定义了函数 main，并没有要求执行 main 函数。

虽然像程序 1.2 那样不将所有语句定义放在函数中也是可以的，但习惯上常定义成 main。 这样做至少有一个好处，那就是一旦导入了模块文件，就可以通过键入 main()来多次执行程序。没有函数的话，就只能通过多次导入模块来执行程序了。

注释

程序中可以使用注释，用于解释变量的含义、函数的功能、模块文件的创建者、程序版 本等等。注释不仅可以帮助他人理解程序，甚至对自己也有帮助理解的作用（试想一下当你 重新拿起几年前写的程序想扩展程序功能时，注释对你的帮助）。

Python 中的注释是以“#”开始的一行，解释器遇见“#”时会自动忽略其后直到行末的 内容。例如我们将上面的 greet()函数存入文件，并加上合适的注释，得到以下程序：

【程序 1.3】eg1_3.py

```
# Author: Lu Chaojun
# eg1_3.py (version 1.0)
def greet():
    print "Hello, Lucy."
    print "How are you?"
    print "Goodbye, Lucy."
greet() # call the function
```

1.4 程序排错

先说一个坏消息：一旦开始写程序，就免不了要出错。程序设计虽然并不难，但无论是 初学编程者还是经验丰富的专业程序员，程序中出现各种错误都是很常见的。

再说一个好消息：计算机（严格说是编译器或解释器）能够帮助我们发现程序中的很多 错误。

在计算机行话中，程序中的错误被称为“臭虫”（bug），而发现并改正错误的过程称为排 错（debug，或称调试）。

程序中的错误大体可分为三种类型：语法错误、运行错误和语义错误。 编程语言和自然语言一样规定了一套语法规则，这些规则定义如何用符号组成形式上正

确的程序。只有符合语法规则的程序才能被计算机执行，语法不正确的程序根本就无法通过编译器或解释器的检查，更谈不上正确执行了。自然语言中的语法比较宽松，犯点语法错误一般不会影响交流，就像有人说的：研表究明，汉顺字序并不定一影阅响读。与自然语言不同，编程语言的语法是非常严格的，任何一点语法错误（例如少了个逗号）都会导致程序无法执行。初学一门编程语言的时候，肯定会出现很多语法错误，但随着对语言的熟悉和经验的增加，语法错误会越来越少。例如：

```
>>> 3 + 4 *
SyntaxError: invalid syntax
```

显然，乘法运算符需要两个运算数，而上面的表达式中未提供足够的数据，因此导致了语法错误。Python 解释器很容易发现语法错误，并将错误信息打印出来供程序员参考。

当程序通过了编译器或解释器的语法检查，就可以运行了。遗憾的是，程序语法正确并 不能保证程序执行成功，因为有很多仅在程序执行时才会出现的错误，这种运行错误也称为 异常（exception）。例如，如果程序中有一条执行除法运算的语句，那么在运行时就有可能发 生除数为 0 的错误，这种错误在编译阶段无法发现，因为除法算式是符合语法的。例如：

```
>>> def f():
    x = 2
    print 10 / x
    x = x - 2
    print 10 / x
>>> f()
5
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module> f()
File "<pyshell#3>", line 5, in f print 10 / x
ZeroDivisionError: integer division or modulo by zero
```

上面这个函数显然没有任何语法错误，因此能够被 Python 执行，我们也看到了部分执行 结果：10 / 2 = 5 被正确地计算并显示出来。但是当 x 变为 0，再次计算 10 / x 时发生了运行 错误 ZeroDivisionError。

语义错误也称逻辑错误，是指程序在程序逻辑上出错，根本不是预定的功能。语法错误 和运行错误都可以被计算机检查出来，程序员根据计算机的报错信息可以比较容易地找出源 程序中的错误并纠正之。而语义错误可能很难发现。因此，有语义错误的程序往往能够“成 功”执行，不产生任何错误消息，但是程序的结果并不是我们想要的，或者说程序的意义（语 义）错了。例如，下面这个程序试图计算半径为 5 的圆的面积：

```
>>> pi = 3.1416
>>> r = 5
>>> print pi * r
15.708
```

程序的执行没有产生任何错误，但结果根本不是圆的面积。

由于得不到 Python 解释器的帮助，查找语义错误往往是非常恼人的。一般来说我们需要 从头到尾仔细审查程序算法，找出可能的错误步骤。一种有用的排错方法是在程序中插入大 量的 print 语句，用来显示计算的中间结果。通过检查中间结果，可以将错误进行精确定位。 这些 print 语句的目的是帮助排错，一旦程序完全正确，再将它们从程序中删除。

排错是程序设计的最重要技能之一。找出程序中的错误一方面很令人头痛，另一方面也 很有乐趣，因为排错过程有点像侦探破案的过程——寻找线索、推断原因、最终定位错误。 如果情况很复杂，对某处代码是否错误不是很肯定，则可以利用试错法来排错：先试着修改 该处代码，然后运行程序看看结果如何。这个过程可以重复进行，直至确定错误为止。

1.5 练习

1. 计算机的主要部件有哪些？工作机制是怎样的？

2. 什么是机器语言、汇编语言和高级编程语言？

3. 高级语言的编译和解释分别是怎样的过程？

4. 什么是计算？

5. 为什么计算机是通用的（即可以应用于各行各业）？

6. 算法和程序有何异同？

7. 计算思维建立在什么原则之上？

8. 请回顾你在玩扑克牌时，抓牌过程中是如何整理顺序的。

9. 假如我们玩猜数游戏：我心中想好一个 1～100 的自然数让你来猜，猜错的话我会告诉你太大或太小，直至你猜中。为了尽快猜中，你有什么好方法？

10. 你会下棋（围棋、象棋、五子棋均可）吗？下棋时你是如何一次计算多步的？

11. 程序错误有哪几类？

12. 设计你的第一个程序：让计算机跟你打招呼（假设你叫 John）

```
Hello John!
Have fun with Python!
```

分别以交互方式和程序文件方式来执行你的程序。

第 2 章 用数据表示现实世界

第 1 章说过，计算是利用计算机解决问题的过程。待解决的问题可能来自不同领域，因 而具有不同的形式和内容，但从计算的角度看，解决任何问题的过程都是对特定信息进行特 定处理的过程。可见，计算涉及到两样东西：信息和对信息的处理过程。因此实现计算的程 序相应地也要做两件事情：第一，用特定数据类型和数据结构将信息表示出来；第二，用控 制结构将信息处理过程表示出来。

本章是关于信息表示的，主要介绍一些简单数据类型以及如何处理这些简单数据。复杂 数据的表示将在第 5、6、7 章介绍。至于信息处理过程的表示，则是第 3、4 章的内容。

2.1 数据和数据类型

2.1.1 数据是对现实的抽象

利用计算机解决现实问题时，首先需要将问题所涉及的信息和处理过程表示成计算机能

够接受的形式。如何建立现实问题的计算机表示呢？显然不能像照相机那样，追求将现实景物事无巨细地复制到胶卷或 CCD 上，因为一个复杂问题所涉及的信息非常多，完全表示它们几乎是不可能的。就拿照相来说，胶卷上的图像能表示事物的重量和人物间的亲属关系吗？一方面无法表示问题的所有信息，另一方面也没有必要建立问题的完美表示。问题所涉信息中一般只有部分信息与问题的解决有关，因此只需对现实问题进行抽象，抽取一部分与问题求解有关的信息进行表示，而忽略那些与问题求解不相干的信息。可见，抽象是对问题进行简化的重要手段。

读者对“数据”这个术语肯定不陌生，但若要问究竟什么是数据，恐怕多数人都很难准 确回答。在现实生活中，数据大体上是指各种事实或数值，当今使用更多也更时髦的术语是 “信息”。而在计算领域，我们将现实世界中的事实或信息用编程语言提供的符号化手段进 行表示，这种符号化表示称为数据（data）。

假设我们测得当前气温是摄氏 35 度，显然这是现实世界的信息。为了用计算机解决某 个涉及温度的问题，就需要将温度信息用计算机能接受的方式表示出来。例如可以用整数 “35”表示，也可以用整数“95”表示（假如采取华氏温标的话），还可以用文本“摄氏 35 度”表示。这几种表示都是编程语言支持、计算机能理解的形式，具体采用哪种形式来表示 温度取决于程序打算对温度数据进行什么处理，通常都会以数值数据来表示温度信息，以便 对温度进行数学计算。

又如，假设我们要用计算机解决学生信息管理的问题，就需要在计算机中用数据来表示 现实世界中的学生。这个数据是对学生的抽象，例如可能包括学生的学号、姓名、年龄等信 息，而不大可能包括学生的发型、是否追星族等与问题求解无关的信息。因此，现实中的学 生张三可能最终被抽象表示成计算机中的数据(2013001,张三,18)，参见图 2.1。

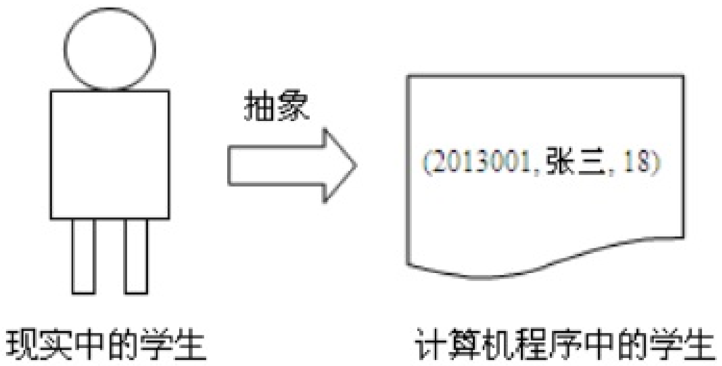


图 2.1 数据是对现实的抽象

总之，为了用计算机解决一个问题，必须先对该问题进行抽象，定义问题在计算机中的

数据表示。数据表示的选择，必须依据将对数据施加的操作来考虑，以便将来能够方便、高效地处理数据。

2.1.1 常量与变量

在程序中如何指明要处理的数据？所有编程语言都提供两种指明数据的方式：第一，直接用字面值（literal）表示数据，即从文本字面上即可看出是什么数据，这种数据是不会改 变的常量；第二，将数据存储在一个变量中，以后用该变量来指代数据。

回顾第 1 章中我们所写的第一个程序：

```
>>> print "Hello World!"
```

其中"Hello World!"就是以字面值的形式指明 print 命令要操作的数据。我们也可以这 样做：

```
>>> s = "Hello World!"
>>> print s
Hello World!
```

这里先将数据"Hello World!"存储在变量 s 当中，然后通过引用 s 来指明 print 要操 作的数据。

又如，3.1416 也是字面值，看到这串文本就知道它表示一个数值。我们可以直接处理 这个字面值，也可以将它存储在变量中并通过引用变量来指代此数值。

```
>>> print 3.1416
3.1416
>>> p = 3.1416
>>> print p
3.1416
```

字面值的意义是不可改变的，而变量的意义（即变量存储的值）是可以改变的。例如， 我们接着上面的语句继续操作数据 p：

```
>>> p = 2.71828
>>> print p
2.71828
```

这里我们将变量 p 的值改成了 2.71828，因此 p 所表示的数据被改变了。 在程序中直接使用字面值通常不是好的做法，因为这会导致程序缺乏一般性，即只适用

于特定计算。如果要将程序应用于其他数据的计算，则必须修改程序中的字面值，这是很不方便的。显然，使用变量可以使程序具有一般性，因为只要为变量赋予不同的值，程序就可以对不同数据进行处理。

变量只是一个“占位符”，必须用具体数据赋值后才有意义。正如我们已经多次见到的， 赋值语句的语法形式是：

```
<变量> = <表达式>
```

其中等号表示赋值，等号左边是一个变量，右边是一个表达式（由常量、变量和运算符构成）。 Python 首先对表达式进行求值，然后将结果存储到变量中。如果表达式无法求值，则赋值 语句出错。一个变量如果未赋值，则称该变量是“未定义的”。在程序中使用未定义的变量 会导致错误。例如：

```
>>> print q
Traceback (most recent call last):
File "<pyshell#21>", line 1, in <module>
print q
NameError: name 'q' is not defined
```

并行赋值

与许多编程语言不同，Python 语言允许同时对多个变量赋值，例如：

```
>>> x,y = 1,2
>>> x
1
>>> y
2
```

这种形式的赋值语句使得交换两个变量的值的任务变得轻而易举：

```
>>> x,y = y,x
>>> x
2
>>> y
1
```

而在其他编程语言中为了交换两个变量 x 和 y 的值，必须借助于一个临时变量，执行三 条赋值语句：

```
temp = x x = y
y = temp
```

2.1.2 数据类型

在前面的例子中出现了两种不同形式的数据值，即"Hello World!"和 3.1416，这 告诉我们计算机所处理的数据是多种多样的，或说具有不同的数据类型。注意，在计算机硬 件层次上并没有什么数据类型的概念，因为所有数据在计算机底层都是二进制序列。只是到 了高级编程语言层次，才提供了数据类型概念。

为了更精细、更准确地表示现实世界的信息，编程语言提供了多种数据类型（data type） 来区分不同种类的数据。早期的数据类型概念相当于定义一个合法值的集合，如果一个数据（变量）是 T 类型的，就意味着该数据（变量）只能取 T 的值集合中的值。后来，数据类 型概念不仅要考虑合法值是什么，而且还要考虑对这些合法值的合法操作是什么。因此，每 一种数据类型由两部分构成：全体合法的值（value）以及对这种值能执行的各种操作（operation，或称运算）。 例如，从小学数学开始，我们逐步认识了自然数、整数、实数、复数等数值集合，并且学会了各数集上的加减乘除等运算方法。除了数值，我们还学习了向量，并且知道向量的运 算方式和数值是不一样的。

为什么要将数据划分为各种数据类型？数据类型决定了合法的数据操作，不合法的操作 将导致程序错误。因此，数据类型的重要作用是通过类型检查来发现程序中的错误，例如企 图将一个人的姓名乘以他的年龄显然是没有意义的。如果不将现实世界的信息在计算机中分 门别类地表示，计算机就无法帮助我们发现像姓名乘以年龄这样的无意义操作。这些错误将 在程序运行的时候暴露出来，导致程序崩溃。有了数据类型的概念，编译器或解释器就能早 早发现程序中的这种错误，使程序在运行之前就有机会修改错误。在这个意义上，数据类型 起到了“量纲分析”①的作用。

> ① 物理量的量纲可用来分析、检验几个物理量之间的关系，这种方法称为量纲分析（dimensional analysis）。

学习利用计算机解决实际问题，一般都是从学习各种数据类型入手。学习每一种类型时， 应该考虑两个问题：该类型的值可以用来表示现实世界的什么信息？现实世界的信息处理任 务可以用该类型的什么操作实现？

编程语言中一般都预定义了一些基本数据类型，或称内建（built-in）类型，如 Python 语言中的数值（int、long 和 float）、字符串（str）、布尔值（bool）、列表（list）、元组（tuple）、 字典（dict）等。此外，编程语言还允许在基本数据类型的基础上构造更复杂的数据类型。

2.1.3 Python 的动态类型*

如果将计算机内存单元比喻成宾馆的房间，那么编程语言中的变量可以理解成这些房间的“门牌标识”。将一个数据存入变量，实际上是存入该变量所标识的内存单元；而访问一 个变量，当然就是访问该变量所标识的内存单元中的数据。

绝大多数编程语言中对变量的使用有严格的类型限制，一个变量固定作为某内存单元的 标识，并且该单元只能存储特定类型的数据。这就好比宾馆的房间分为客房、员工房和工作 间等，客房又分单人间、双人间和套房等，每个房间有固定的门牌号，不同人员只能进入规 定的房间。如果一个变量预先声明为只能存入数值数据，那就不能将字符串存进该变量；一 旦发生存入的数据与预先声明的类型不一致的情况，程序即出错。我们称这种编程语言是静 态类型化的。

然而，Python 语言采用的是另一种技术——动态类型化。在 Python 中，变量并不是某 个固定内存单元的标识，也就不需要预先定义变量的类型。事实上，Python 变量是对内存 中存储的某个数据的引用（reference），这个引用是可以动态改变的。变量的类型就是它所 引用的数据的类型，对变量的每一次赋值，都可能改变变量的类型。还是用宾馆的比喻，这 就好比宾馆房间没有固定的门牌号码，某个门牌号 N 今天可以挂在单人间门上，明天又可 以换到总统套房的门上。于是 N 今天是单人间类型，明天又是套房类型，总之类型是动态 确定的。

例如，执行下面的赋值后，Python 在内存中创建数据 123，并使变量 x 指向这个数据， 因此可以说 x 的类型现在是整数类型。

```
>>> x = 123
>>> print x
123
```

如果进而执行下面的赋值语句，则 Python 又在内存中创建数据"Hello"，并使 x 改为指向 这个字符串数据，因此 x 的类型现在变成了字符串类型。参见图 2.2。

```
>>> x = "Hello"
>>> print x
Hello
```

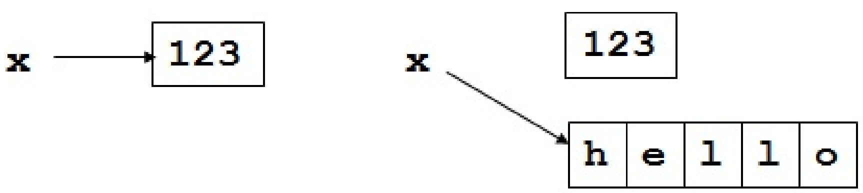


图 2.2 变量的动态类型化

顺便说一下，当 x 从 123 转而指向"Hello"后，数据 123 就变成了无人使用的“垃圾 数据”（除非还有别的变量引用它），Python 会回收垃圾数据的存储单元，以便提供给别的 数据使用，这称为垃圾回收（garbage collection）。读者可以思考一下，如果没有垃圾回收，会造成什么后果？

2.2 数值类型

自然界的事物都具有数量属性，由此抽象出了数的概念，所以数值几乎无处不在。计算 机曾被认为是数值计算的机器，并且至今数值计算仍然是计算机的重要应用领域。事实上， 从最底层来看，计算机也只会对二进制数值进行操作。高级编程语言中的种种数据类型及其 操作，最终都要转化成底层的二进制数值计算。

最常用的数值类型包括整数和浮点数类型。

2.2.1 整数类型 int

整数就是没有小数部分的数值，分为正整数、0 和负整数。Python 语言提供了类型 int 用于表示现实世界中的整数信息，如班级里的人数、人的年龄、乒乓球比赛每方的得分等等。

基本数据类型的值都可通过字面值（literal）的形式表示出来，即以字面形式表现值。 整数类型的字面值表示形式和我们在现实世界中的写法一样，例如下列都是合法的整数：

```
123 -456 0
```

注意，整数字面值是不能包含小数点的，即使小数点后面什么都没有！读者也许会觉得 这句话很奇怪，因为在数学中从没见过一个数包含小数点但小数点后面啥也没有的情形。然 而，在 Python 中确实允许以下形式的字面值：

```
123. -456. 0.
```

但它们都不是整数！事实上，以上三个数分别等于 123.0、-456.0 和 0.0，它们属于后 文即将介绍的浮点数类型。

Python 语言为整数类型提供了通常的数学运算，运算符及其含义如表 2.1 所示：

运算符	含义
+	加
-	减
*	乘
/	除
**	乘方
%	取余数
abs()	取绝对值

例如：

```
>>> 23 + 45
68
>>> 56 - 12
44
>>> 8 * 2
16
>>> 11 / 3
3
>>> 8 ** 2
64
>>> 18 % 5
3
```

表 2.1 整数运算符

```
>>> abs(-8)
8
```

可见，计算机实现的整数运算基本上和我们在数学课上所学的一样，除了一个例外—— 除法。由于例中的 11/3 是整数类型上的除法，运算结果仍然在整数类型当中，所以 Python 将商的小数部分直接舍弃了（未作四舍五入！），从而结果为 3。在程序中，本来希望得到精确的除法结果，但因被除数和除数都是整数，导致结果误差过大甚至出错，这是初学 Python 编程的人很容易防错误的地方。要说明一下，表 2.1 中的 abs()并不是运算符，而是 Python 的内建函数，这里只是为了方便而将它列在了表中。

除了上面这些运算符，Python 还提供了一些运算符与变量赋值结合起来的表示法。例 如，在程序设计中经常用到一个变量递增的操作：x = x + 1。注意，这个式子在数学中 是不成立的，因为一个数不可能“等于”该数加 1。但在编程语言中这是一个完全合法的赋 值语句，它的含义是：将变量 x 所指向的值加 1，并将计算结果重新赋值给 x。鉴于这个操 作频繁使用，Python 和某些其他语言提供了一种简写形式：x += 1。请看例子：

```
>>> x = 123
>>> x += 1
>>> print x
124
```

还有其他一些类似的简写形式，参见表 2.2。

普通形式	简写形式
x = x + y	x += y
x = x - y	x -= y
x = x * y	x *= y
x = x / y	x /= y
x = x % y	x %= y

表 2.2 赋值与运算结合

int 类型的局限性

在第 1 章中我们说过，计算思维是建立在计算机的能力和限制之上的。现在我们来讨论 整数类型的一个限制。

int 类型只是数学中的整数集合 I 在计算机中的表示，而一个事物和该事物的一种表示 之间未必可以划等号。事实上，类型 int 只表示了 I 的一个子集，I 是无穷集合，而 int 是有穷的。这是为什么呢？

在计算机底层，整数一般都是用特定长度的二进制数表示的。至于具体长度是多少，取 决于 CPU 的设计。目前个人计算机上多采用 32 个二进制位（bit，比特）的长度来表示整数， 故 Python 语言中的 int 类型就是 32 比特长度的整数值。利用一点排列组合知识，容易推 知：一个比特有两种可能的状态（0、1）， 两个比特有四种可能的状态（00、01、10、11）， 三个比特有八种状态（000、001、010、011、100、101、110、111），…，32 个比特有 232

种可能的状态。用这 232 种状态显然只能表示 232 个整数，考虑到整数有正负，计算机底层将这 232 个状态的一半用于表示非负整数，另一半用于表示负整数，从而类型 int 实际上是由-231～231-1 之间的所有整数构成的集合①。

我们已经了解，数据是现实世界信息在计算机中的抽象，根据数据值的种类和操作的不 同而划分成不同数据类型。一般来说在逻辑层次上理解和使用数据类型就够了，不需要进一 步了解这些抽象在计算机底层的物理表示。然而，如果能对数据类型的底层表示方法有所了解，可以使数据和程序设计更好地建立在机器的能力和限制之上。

> ① 有的语言还支持用 32 比特表示 0～232-1 的无符号整数。

2.2.2 长整数类型 long

如果在计算过程中出现超出 int 范围的整数怎么办？我们来看一个例子：

```
>>> 123456789 * 10
1234567890
>>> 123456789 * 18
2222222202L
```

注意观察第二个表达式的结果——2222222202 的后面有个“L”。我们对此解释如下：第 一个表达式的计算没有问题，因为 1234567890 处于 int 类型范围之内；而第二个表达式 的计算结果 2222222202 已经超出了 int 的范围，Python 对此问题的处理办法是将该结果 转化成另一种整数类型，即长整数①。

长整数类型 long 的值在计算机内的表示不是固定长度的，只要内存许可，长整数可以 扩展到任意长度。因此，使用长整数类型几乎能表示无限的整数。长整数类型的字面值必须 加后缀“L”或“l”，这是 long 类型的标志，Python 看到这个标志就会按长整数的存储方 式来存储。因此，5 和 5L 虽然都表示整数 5，但它们在计算机内部具有完全不同的表示， 分属于不同的类型。为了证实这一点，我们用 Python 中检查表达式类型的函数 type()来 检查 5 和 5L 的类型，结果如下：

```
>>> type(5)
<type 'int'>
>>> type(5L)
<type 'long'>
```

long 类型和 int 类型除了内部表示不同，运算规律是一样的。例如 long 类型同样支 持表 2.1 中的所有运算。下面是两个例子：

```
>>> 2L + 3L
5L
>>> 1234567890987654321L % 123456789L
9L
```

要注意的是，与 int 类型相比，long 类型的运算效率较差。这是因为 int 类型的运 算是 CPU 硬件直接支持的，而 long 类型的运算是用程序实现的。所以，除非有必要，程 序中应当尽量使用 int 类型表示整数信息。

顺便说一下，如果用 print 语句来显示表达式的计算结果，print 会对计算结果进行 一些修饰处理，以使输出更好看。对于长整数，print 会去掉后缀 L，例如：

```
>>> print 2L + 3L
5
```

最后给读者出一道“娱乐题”，将紧绷的“计算思维”放松一下。请思考下面这条语句 的结果是怎么回事？

```
>>> print 2l + 3
5
```

自动类型转换：int 与 long

一般说来，只有同类型的数据才能相互运算。例如，int 数据和 int 数据相互运算， 结果还是 int 类型的数据；long 数据和 long 数据相互运算，结果还是 long 类型的数据。

① 较老版本的 Python 遇到这种情况会报错。

然而，由于 int 和 long 都是整数（只是内部表示不同），所以这两个类型的数据之间相互 运算完全是合理的。问题是，int 数据与 long 数据相互运算的结果是什么类型呢？

为了执行混合类型的两个数据的运算，Python 需要先将它们转换成同一类型。那么是 将 int 转换成 long，还是将 long 转换成 int？一般而言，数据类型转换应当确保不丢失 信息。将 long 数据转化成 int 数据是不安全的，因为 int 的可表示整数范围较小，大整 数无法转换成 int；相反，任何 int 都可以转换成 long。因此，对 int 和 long 混合的 表达式，Python 自动将 int 数据转换成 long 数据之后再运算，运算结果当然就是 long 类型的。例如：

```
>>> 5 * 6L
30L
```

Python 在计算 5*6L 时，先将 5 转化成 5L，再执行长整数的乘法运算，从而得到 30L。 另外，当两个 int 类型的数据进行运算，导致结果超出 int 范围时，较后版本的 Python

也会自动将结果转换成 long 类型的数据。前面我们已经看过这样的例子。

计算是次序的艺术

最后来看一个有趣的例子。如前所述，int 类型所能表示的最大整数是 231 - 1，我们来 计算这个表达式的值：

```
>>> 2 ** 31 - 1
2147483647L
```

奇怪的是，2147483647 明明是在 int 范围之内的整数，怎么会加上了长整数类型的 后缀 L 呢？对此问题，看看 231 – 1 的计算过程就明白了：Python 在计算这个表达式的时候 是先计算 231，然后再减去 1。而在得出中间结果 231 = 2147483648 时已经超出 int 范围了， 计算机只能将此中间结果用 long 类型的整数来表示，接下来的减 1 也就变成了 long 类型 的减法。

那么，有没有办法计算 231 – 1 但是计算结果不带后缀 L 呢？有一个巧妙的迂回策略可 以达到目的，计算过程如下：

```
>>> 2 ** 30 – 1 + 2 ** 30
2147483647
```

看明白了吧，这里用到了简单事实 231 = 230 + 230，从而 231 – 1 = 230 – 1 + 230。在从左向右 计算这个表达式的过程中，所有中间结果都是 int 范围内的值。

这个小例子虽然很简单，但它说明了计算不同于数学的一个特点：计算是紧密依赖于操 作步骤、操作次序的艺术。当一条计算途径行不通，也许改变一下次序就可以解决。而在数 学中，谁也不会认为 231 - 1 和 230 – 1 + 230 之间有什么不同。这验证了我们在第 1 章说过的计算思维的根本原则：计算必须充分利用计算机的能力，避开计算机的限制。建议读者好好体会这种思想。

2.2.3 浮点数类型 float

浮点数就是包含小数点的数，大体对应于数学中的实数集合。现实世界中的职工工资（以 元为单位）、房屋面积（以平方米为单位）、人的身高（以米为单位）、圆周率等在程序中都适合用浮点数表示。

Python 语言提供了类型 float 用于表示浮点数。float 类型的字面值形式与数学中的 写法基本一致，但是允许小数点后面没有任何数字（表示小数部分为 0），例如下列字面值 都是浮点数：

```
3.1415 -6.78 123.0 0\. -6.
```

Python 为浮点数类型提供了通常的加减乘除等运算，运算符与整数类型是一样的（见 表 2.1）。但是，与整数类型不同的是，运算符“/”用于浮点数时，是要保留小数部分的， 例如：

```
>>> 11.0 / 3.0
3.6666666666666665
```

没错，最后一位小数是 5 而不是 6！原因见下面关于浮点数内部表示的内容。 将一个浮点数赋值给变量，则该变量就是 float 类型（实际上是指向一个 float 类型的数据）。例如：

```
>>> f = 3.14
>>> type(f)
<type 'float'>
```

浮点数运算同样可以和变量赋值结合起来，形成如表 2.2 所示的简写形式。

浮点数的能力与限制 浮点数类型能够表示巨大的数值，能够进行高精度的计算。但是，由于浮点数在计算机

内是用固定长度的二进制表示的，有些数可能无法精确地表示，只能存储带有微小误差的近似值。例如，

```
>>> 1.2 – 1.0
0.19999999999999996
```

结果比 0.2 略小。又如：

```
>>> 2.2 – 1.2
1.0000000000000002
```

结果比 1.0 略大。然而，下面这个表达式却计算出了精确结果：

```
>>> 2.0 – 1.0
1.0
```

尽管浮点表示带来的这种微小误差不至于影响数值计算实际应用，但在程序设计中仍然 可能导致错误。例如，万一某个程序中需要比较 2.2 ? 1 是否等于 1.2，那我们就得不到 预期的肯定回答，因为 Python 的计算结果是不相等！请看下面两个比较式：

```
>>> (1.2 – 1.0) == 0.2
False
>>> (2.0 – 1.0) == 1.0
True
```

先解释一下，上例中用到了比较两个表达式是否相等的运算符“==”，另外显示结果出 现了表示真假的布尔值 True 和 False，这些内容在后面布尔类型一节中有详细介绍。从 这个例子我们得到一条重要的经验：不要对浮点数使用==来判断是否相等。正确的做法是 检查两个浮点数的差是否足够小，是则认为相等。例如：

```
>>> epsilon = 0.0000000000001
>>> abs((1.2 – 1.0) - 0.2) &lt; epsilon
True
```

另外从运算效率考虑，与整数类型 int 相比，浮点数类型 float 的运算效率较低，由 此我们得出另一条经验：如果不是必须用到小数，那就应当使用整数类型。

科学记数法

对于很大或很小的浮点数，Python 会自动以科学记数法来表示。所谓科学记数法就是

以“a×10 的整数次幂”的形式来表示数值，其中 1 <= abs(a) < 10。例如，12345 可 以表示成 1.2345e+4，0.00123 可以表示为 1.2345e-3。下面是 Python 的计算例子：

```
>>> 1234.5678 ** 9
6.662458388479362e+27
>>> 1234.5678 ** -9
1.5009474606688535e-28
```

正如 int 不同于整数集 I 一样，Python 的 float 也不同于实数集 R，因为 float 仍 然只能表示有限的浮点数。当一个表达式的结果超出了浮点数表示范围的时候，Python 会 显示结果为 inf（无穷大）或-inf（负无穷）。读者可以做一个有趣但略显麻烦的实验，试 一试 Python 最大能表示多大的浮点数。下面是本书著者所做的实验结果，可以看到，最大 浮点数的数量级是 10308，有效数字部分已经精确到小数点后面第 53 位（Python 在显示结果 时只保留小数点后 16 位），当该位为 6 时是合法的浮点数，当该位为 7 时则超出范围。

```
>>> 1.79769313486231580793728971405303415079934132710037826e+308
1.7976931348623157e+308
>>> 1.79769313486231580793728971405303415079934132710037827e+308
inf
```

顺便说一下，如果读者做这个实验，相信你一定会采用一种快速有效的策略来确定每一 位有效数字，而不会对每一位都从 0 试到 9。例如，当发现 1.7…1e+308 是合法的浮点数， 而 1.7…9e+308 超出了范围，接下去应当检查 1.7…5e+308 的合法性。这种方法就是本 书后面算法设计一章中介绍的二分查找策略。我们在第 1 章说过，计算思维人人皆有、处处 可见，不是吗？

自动类型转换

float 类型与 float 类型的数据相互运算，结果当然是 float 类型。问题是 float 类型能与 int 或 long 类型进行运算吗？

由于整数、长整数和浮点数都是数值（在数学上都属于实数集合 R），因此 Python 允许

它们混合运算，就像 int 可以与 long 混合运算一样。Python 在对混合类型的表达式进行 求值时，首先将 int 或 long 类型转换成 float，然后再执行 float 运算，结果为 float 类型。例如：

```
>>> type(2 + 3.0)
<type 'float'>
>>> type(2 + 3L * 4.5)
<type 'float'>
```

手动类型转换

除了在计算混合类型的表达式时 Python 自动进行类型转换之外，有时我们还需要自己 手动转换类型。这是通过几个类型函数 int()、long()和 float()实现的。例如，当我 们要计算一批整型数据的平均值，程序中一般会先求出这批数据的总和 sum，然后再除以数 据的个数 n，即：

```
average = sum / n
```

但这个结果未必如我们所愿，因为 sum 和 n 都是整数，Python 执行的是整数除法，小数部分被舍弃了，导致结果误差太大。为解决此问题，我们需要手动转换数据类型：

```
average = float(sum) / n
```

其中 float()函数将 int 类型的 sum 转换成了 float 类型，而 n 无需转换，因为 Python 在计算 float 与 int 混合的表达式时，会自动将 n 转换成 float 类型。

要注意的是，下面这种转换方式是错误的：

```
average = float(sum/n)
```

因为括号里的算式先计算，得到的就是整除结果，然后再试图转换成 float 类型时，已经 为时已晚，小数部分已经丢失了。

其实，调用类型函数来手动转换类型并不是好方法，我们有更简单、更高效的做法。如 果已知的数据都是整数类型的，而我们又希望得到浮点类型的结果，那么我们可以将表达式 涉及的某个整数或某一些整数加上小数点，小数点后面再加个 0，这样整数运算就会变成浮 点运算。例如求两个整数的平均值：

```
>>> x = 3
>>> y = 4
>>> z = (x + y) / 2.0
>>> z
3.5
```

例中我们人为地将数据个数 2 写成了 2.0，这样就使计算结果变成了 float 类型。 当然，在将浮点数转换成整数类型时，就没有这种简便方法了，只能通过类型函数来转换。例如：

```
>>> int(3.8)
3
>>> long(3.8)
3L
```

可见，float 类型转换成 int 或 long 时，只是简单地舍去小数部分，并没有做四舍五入。 如果希望得到四舍五入的结果，一个小技巧是先为该值（正数）加上 0.5 再转换。更一般 的方法是调用内建函数 round()，它专门用于将浮点数转换成最接近的整数部分。不过舍 入后的结果仍然是 float，为了得到 int 类型的数据还需要再用 int()转换。例如：

```
>>> round(3.14)
3.0
>>> round(-3.14)
-3.0
>>> round(3.5)
4.0
>>> round(-3.5)
-4.0
>>> int(round(-3.14))
-3
```

2.2.4 数学库模块 math

对于数值类型，除了加减乘除等基本运算之外，Python 还以“数学库”的形式提供了 很多数学函数，以丰富编程所需的数学计算手段。所谓“库”其实是专业程序员编写的 Python 模块，其中定义了很多有用的函数，应用程序可以使用库中的函数，就好像是应用程序自己 定义的函数一样。

为了使用数学库 math 中的函数，在程序中首先要用 import 语句导入 math 模块：

```
import math
```

导入一个模块的效果相当于将该模块中定义的函数代码拷贝到我们自己的程序中，从而当调用库函数的时候，Python 知道这些函数是在哪里定义的。

例如，math 库中定义了一个函数 sqrt()，其功能是计算一个数的平方根。导入了 math

之后，可以通过下面的方式来使用这个函数：

```
>>> import math
>>> math.sqrt(16)
4.0
```

其中 math.sqrt()这种表示法就相当于说“调用模块 math 中的 sqrt 函数”，导致 Python 去 math 库（已导入）中查找 sqrt 函数并调用之。顺便说一下，即使没有 math 库，Python 也能计算平方根——不要忘了乘方运算符**，平方根其实就是 0.5 次方。

其实还有另一种导入模块中函数定义的方式，形如：

```
from math import sqrt
```

这条语句的含义是：从 math 模块导入 sqrt 函数的定义。这种导入方式的好处是，将来调 用 sqrt 的时候不必使用模块名作为前缀，而可以直接调用 sqrt。例如：

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

如果希望导入 math 模块中的所有定义，而非仅仅导入 sqrt 函数，则可使用如下形式：

```
from math import *
```

此处的星号表示“所有定义”的意思。

表 2.3 给出了 math 库中定义的一些数学函数和常数。

Python	含义
pi	常数pi（近似值）
e	常数 e（近似值）
sin(x)	正弦函数
cos(x)	余弦函数
tan(x)	正切函数
asin(x)	反正弦函数
acos(x)	反余弦函数
atan(x)	反正切函数
log(x)	自然对数（以 e 为底）
log10(x)	常用对数（以 10 为底）
exp(x)	指数函数 e^x
ceil(x)	大于等于 x 的最小整数
floor(x)	小于等于 x 的最大整数

表 2.3 math 库中的常用函数

2.2.5 复数类型 complex*

Python 语言还有内建的 complex 类型用于表示复数。在数学中，任一复数可表示为 a + bi，a 称为实部，b 称为虚部。而在 Python 中，complex 类型的字面值形式是(a+bj)，在 不会产生误解的情况下括号也可以省略。注意虚数符号是 j 或 J，而不是数学中用的 i。

对复数类型同样可以执行表 2.1 中的所有运算。有一点不同的地方是，abs()对复数来 说是计算复数的模数。例如：

```
>>> c1 = 2 + 4j
>>> c2 = 7 + 6j
>>> print c1 + c2
(9+10j)
>>> print c1 – c2
(-5-2j)
>>> print c1 * c2
(-10+40j)
>>> print abs(c1)
4.472135955
```

另外可以通过 x.real 和 x.imag 来分别获得复数 x 的实部和虚部，结果都是 float

类型。例如接着上面的例子继续执行：

```
>>> c1.real
2.0
>>> c2.imag
6.0
```

2.3 字符串类型 str

计算机的早期应用主要是科学计算，处理的都是数值。如今，计算机已经大量地应用于 各种文本数据的处理，例如企业信息管理、文本编辑器、搜索引擎等等。文本数据在程序中 是用字符串类型表示的。

字符是计算机中表示信息的最小符号，常见的大小写字母、阿拉伯数字、标点符号等都 是字符。除了这些看得见的“可打印字符”，还有一些看不见的“控制字符”，例如回车、换 行、退格等等。

字符串是由字符组成的序列，在程序中作为被处理的数据。字符串数据在现实世界中是 非常普遍的，例如人的姓名、家庭地址、身份证号码等等都是字符串数据。

2.3.1 字符串类型的字面值形式

由于计算机程序本身要用字符序列来表示，因此程序中的命令、变量名、字面值、标点

符号等等都是字符组成的序列，但它们是程序构件而不是数据。这就带来一个问题：如何区分程序中的某一个字符序列到底是字符串数据还是程序构件？几乎所有编程语言都采用了 加引号的方法来解决这个问题：字符串数据必须用一对引号括起来。

Python 语言提供了字符串数据类型 str，并且在表示字符串数据方面比其他语言更灵 活。在 Python 中，字符串的字面值有四种形式：

（1）用单引号括起来的字符串，例如

```
>>> 'a string enclosed in single quotes'
'a string enclosed in single quotes'
```

（2）用双引号括起来的字符串，例如

```
>>> "a string enclosed in double quotes"
'a string enclosed in single quotes'
```

（3）用三个单引号括起来的字符串，例如

```
>>> '''a multiple-line string enclosed in
triple quotes'''
'a multiple-line string\nenclosed in\ntriple quotes'
```

（4）用三个双引号括起来的字符串，例如

```
>>> """a multiple-line string enclosed in
triple double-quotes"""
'a multiple-line string\nenclosed in\ntriple double-quotes'
```

用单引号或双引号括起来的字符串必须在一行内表示，是程序设计中最常用的形式。而 用三个单引号或三个双引号括起来的字符串可以是多行的，主要用于一个特殊用法——文档 字符串（docstring），具体用法在略过。

字符串可以存储在变量中，从而得到字符串类型的变量。例如：

```
>>> s = "Hello"
>>> type(s)
<type 'str'>
```

用单引号还是双引号来界定字符串并没有差别，Python 之所以提供这两种表示法，是 为了能更方便地表示某些字符串。例如，如果使用双引号作为界定符，而我们的文本数据是

```
He said, "OK".
```

即字符串数据本身使用了双引号这个字符，那么如下形式的字符串数据

```
"He said, "OK"."
```

显然要出问题，因为 Python 在从左向右读这个字符序列的时候，会将"He said, "解释成 一个字符串数据，然后又因为无法解释后面的字符序列 OK"."而导致出错。为了避免出现 这样的问题，我们可以使用单引号来界定字符串，从而得到

```
'He said, "OK".'
```

Python 对这个字符串完全可以给出正确解释：看到第一个单引号，就知道开始了一个字符串，接下去的字符（包括双引号）都是字符串的组成部分，直至遇见第二个单引号为止。

类似地，如果文本数据中出现了单引号字符，那么我们可以使用双引号作为字符串界定 符，如"Tom's World"之类。较真的读者马上会联想到：那万一文本数据中既有单引号又 有双引号怎么办？例如：

```
He said, "I'll do it".
```

这种情况下用单引号或双引号都会出错。Python 的解决方法是使用转义字符“\”，例如上 面这个文本数据在程序中可以用如下字符串表示：

```
"He said, \"I'll do it\"."
```

其中\"使得双引号不再按界定符的意义解释，而是转变为普通的双引号意义。举一反三， 显然下面这种形式的字符串也是正确的：

```
'He said, "I\'ll do it".'
```

因为用单引号作为字符串界定符，所以字符串内部的单引号要用转义字符“\”来转变意义。

2.3.2 字符串类型的操作

在实际应用中，对字符串最常用的操作是访问字符串中的个别字符。Python 语言为字

符串类型提供了索引操作，可以用来访问字符串内部的任意组成字符。 字符串是字符序列，每个字符在序列中的位置都由一个从 0 开始的整数编号指定，这个

编号称为位置索引。因此，第一个位置的索引是 0，第二个位置的索引是 1，依此类推。通过索引我们可以指定字符串中的任意位置，从而可以访问该位置上的字符。下面是通过索引操作访问字符串内容的一般形式：

```
<字符串>[<数值表达式>]
```

数值表达式的值就是位置索引，整个索引操作的返回结果就是索引位置上的字符。例如：

```
>>> s = "Good morning!"
>>> s[0]
'G'
>>> s[12]
'!'
>>> i = 8
>>> s[i+4]
'!'
```

注意，在长度为 n 的字符串中，最后一个字符的索引位置是 n-1。初学者很容易犯的一 个错误是：因为字符串 s 的长度为 12，所以通过 s[12]来访问其最后一个字符。务必记住， 计算机科学和程序设计中，习惯是从 0 开始计数。

Python 还支持从后往前的索引方式：索引-1 代表倒数第一个位置，索引-2 代表倒数第 二个位置，依此类推。利用这个表示法，无需知道字符串长度即可访问最后一个字符：

```
>>> s[-1]
'!'
```

以上是通过索引操作访问字符串中的单个字符，Python 也支持通过索引操作来访问字 符串的子串，方法是指定字符串的一个索引区间。这种操作也称为切分。切分操作的一般形 式是：

```
<字符串>[开始位置:结束位置]
```

其中开始位置和结束位置都是 int 类型的表达式，含义是返回字符串中从开始位置到结束位置（不含结束位置！）的一个子串。开始位置和结束位置是可选的，在没有指定的情况下 Python

默认开始位置为 0，结束位置为 n。承接上面的例子继续进行如下切分操作：

```
>>> s[0:3]
'Goo'
>>> s[5:13]
'morning!'
>>> s[:10]
'Good morni'
>>> s[5:]
'morning!'
>>> s[:]
'Good morning!'
>>> s[2:-2]
'od mornin'
```

除了索引操作，字符串类型还支持字符串的合并（+）、复制（*）、子串测试（in）操作，并提供一个求字符串长度的内建函数 len()。其中子串测试返回一个布尔值（True 或 False）， 关于布尔类型参见 2.4 节。例如：

```
>>> "Good" + "Bye"
'GoodBye'
>>> 2 * "Bye" 'ByeBye'
>>> "ok" in "cook"
True
>>> len("Good"*3 + 2*"Bye")
18
```

在应用程序中有时也许会希望修改一个字符串，正如现实世界中有人去派出所修改自己 的名字一样。利用索引机制似乎很容易实现修改字符串的功能，例如下面的语句试图将 "Tom"改成"Tim"：

```
>>> name = "Tom"
>>> name[1] = "i"
```

但很遗憾，Python 中的字符串类型的值是不能修改的！上述操作将导致如下结果：

```
Traceback (most recent call last):
File "<pyshell#25>", line 1, in &lt;module&gt; name[1] = "i"
TypeError: 'str' object does not support item assignment
```

其中最后一行的意思是：str 类型的数据不支持对其成员的赋值。name[1]是字符串"Tom" 的第 2 个成员，因此不能对其进行赋值！

最后，我们将以上介绍的各种基本字符串操作整理成表 2.4，以方便查阅。

字符串操作	含义
[]	索引操作
[:]	切分操作
+	合并字符串
*	复制字符串
len(<字符串>)	字符串长度
<字符串 1> in <字符串 2>	子串测试

表 2.4 字符串操作

2.3.3 字符的机内表示

和数值一样，字符在计算机内部也是用二进制数表示的，这个二进制数称为该字符的编码。于是，字符串在计算机内自然就用二进制数的序列表示。可以推知，对字符和字符串的 所有操作，实质上都是对二进制数的运算。我们在屏幕上看到各个字符有各自的形状，这只 是计算机的显示系统将字符的编码映射到特定屏幕像素组合的结果。

表示每个字符的二进制编码具体等于几并不重要，我们可以用(1111)2 表示字符 A，也可 以用(0000)2 表示字符 A，这不会带来什么本质的不同，事实上只要确保不同字符有不同的 编码即可。但是，为了在不同计算机之间能够交换信息，避免发生一台计算机上的字符 A（假设编码是(0000)2）传给另一台计算机后被解释成字符 B（假设(0000)2 在这台机器上恰 好是 B 的编码），我们需要统一字符编码。基于这个思想，人们制定了字符集编码标准—— 定义所支持的字符集以及每个字符的二进制编码。

由于计算机是美国人发明的，所以较早出现的一个编码标准是根据美国的使用情况制定 的标准，称为 ASCII①。这个标准也是最重要的，几乎所有计算机都支持 ASCII 的字符编码。ASCII 使用一个字节的 7 位二进制位来表示字符（最高位恒为 0），这样就只能支持 27 = 128 个字符，各字符的编码如果用十进制表示就是 0～127。ASCII 所定义的字符包括大小写英 文字母、阿拉伯数字、标点符号、空格、回车、换行等，它们分为可打印字符和控制字符两 类。

> ① American Standard Code for Information Interchange 的首字母缩写。

Python 中提供了两个与字符编码有关的函数：ord()函数用于从字符得到其编码，chr() 函数用于从编码得出对应的字符。例如：

```
>>> ord('A')
65
>>> ord('a')
97
>>> ord('8')
56
>>> ord(' ')
32
>>> chr(64)
'@'
>>> chr(10) '\n'
>>> chr(13)
'\r'
```

对此例有几点说明：第四个例子是求空格字符的编码（32）；第六个例子说明编码 10 对应的字符可以用转义字符\n 表示，它其实就是换行字符；第七个例子说明编码 13 对应的字符可以用转义字符\r 表示，它其实就是回车字符。换行和回车都是控制字符的例子，控制字符 不像字母数字那样有可打印、显示的形状，但在程序中可以用转义字符来表示某些控制字符。

ASCII 编码的一个问题是支持的字符太少，对美国人来说够用，但对其他国家来说远远 不够。因此产生了各种对 ASCII 的扩充标准。例如针对欧洲语言的 Latin-1 标准将一个字节 的最高位也用上，从而在 ASCII 的基础上增加了 128 个字符。

中国的汉字也是字符，并且数量很大，用一个字节编码是远远不够的。较早的国家标准 GB2312 采用两个字节来对汉字编码，共定义了 6763 个汉字。后来产生了 GBK 规范，仍然用两个字节编码，但支持 2 万多个汉字。最新的国家标准是 GB18030，它最多可用四个字 节编码，支持 7 万多个汉字。

为了将全世界的字符编码统一起来，国际标准化组织 ISO 制定了一个庞大的字符编码 标准 Unicode。Unicode 最多用四个字节的编码，因此可以囊括地球上所有语言所用到的所 有字符，目前已经得到广泛支持。较新版本的 Python 语言（包括 2.7 版）都支持 Unicode。 下面我们举例说明 Python 对非 ASCII 字符的处理方法。最简单的方法是使用 Unicode 字符串。Python 语言中，在字符串前面加个前缀 u 就表示 Unicode 字符串，其中可以使用

任意 Unicode 字符。例如：

```
>>> print u'A\xc4B'
AÄB
```

在这个例子中，字符串由三个字符构成：头尾两个字符分别是 A、B，可以从键盘直接 输入；中间的字符是 Latin-1 字符集中的字符 ?，无法从键盘直接输入，但可以通过输入十 六进制编码（即 c4，另外\x 是十六进制数的标志）的方式来输入。

再看汉字的例子：

```
>>> '汉'
\xba\xba
>>> print '汉'
汉
>>> print '\xba\xba'
汉
```

从第一条语句可以看出，我们输入的“汉”字在机器内部被表示成了两个字节的编码，该编码按十六进制表示等于 baba，亦即 GBK 规范中“汉”的编码①。接下来两条 print 语句 表明，字符“汉”和编码“\xba\xba”作用是一样的。

如果需要将汉字和 ASCII 字符、Latin-1 字符等混合在一起构成字符串，那就只能用 Unicode 字符串。例如，“汉”在 Unicode 中的编码是 6c49，在 Unicode 字符串中可以用\u6c49 代表“汉”。结合前面的例子，读者应能理解下面这条语句的结果：

```
>>> print u'A\u6c49\xc4B'
A 汉 隈
```

如果希望 Python 程序能够处理包含汉字的字符串，用 Unicode 字符串是最可靠的做法。

具体细节在此从略。

2.3.4 字符串类型与其他类型的转换

应用程序中有时需要将字符串类型的数据转换成其他数据类型，或者相反。下面介绍Python中如何实现这些功能。

首先看函数 eval()。eval 函数接收一个字符串，并将该字符串解释成 Python 表达式 进行求值，最终得到特定类型的结果值；如果字符串无法解释成合法的 Python 表达式则报 错（如语法错误、未定义变量错误等）。例如：

```
>>> eval("3.14")
3.14
>>> eval("1+2*3/4%5")
2
>>> eval("a+1")
Traceback (most recent call last):
File "<pyshell#34>", line 1, in <;module> eval("a+1")
File "<stringv", line 1, in <module> NameError: name 'a' is not defined
>>> a = 10
>>> eval("a+1")
11
>>> eval("a &gt; 8 and True")
True
>>> s = "Hello"
>>> eval("s + 'World'")
'HelloWorld'
```

最后一个例子表明 eval 也可以对字符串类型的表达式求值，当然这没什么意义，eval 的主要用途是对字符串形式的数值表达式求值。例如从键盘输入一个表达式或者从一个文本 文件中读取一个表达式的场合，都需要用 eval 来求值。

如果字符串的形状符合某种类型的字面值的形式，则可以直接用 int()、long()、 float()、bool()等来转换类型。这里 bool 是布尔类型，详见 2.4 节。如：

```
>>> int("123")
123
>>> long("123")
```

> ① 这是 Windows XP 平台（默认用 GBK）下的结果。不同平台会有不同编码。

```
123L
>>> float("123")
123.0
>>> bool("True")
True
```

如果希望将其他类型的值转换成字符串类型，可以使用 str()函数。例如：

```
>>> str(123)
'123'
>>> a = 123.4
>>> print str(a) + "567"
123.4567
```

注意最后这个例子用到了字符串的合并运算。如果不转换变量 a 的类型，Python 就会将“+”解释成数值加法，但后一个操作数是字符串而非数值，结果即导致错误。

2.3.5 字符串库 string

和数学库 math 一样，Python 还提供了字符串库 string，以支持更复杂的字符串操作。 为了使用 string 中的函数，必须先导入该模块。回忆一下，模块有两种导入方式：

```
import string
from string import *
```

它们的区别在于调用函数时是否需要加上模块名作为前缀。 模块 string 中的一些常用函数如下表所示：

函数	含义
capitalize(s)	将 s 的首字母改成大写
capwords(s)	将 s 中的每个单词的首字母改成大写
center(s,width)	将 s 扩展到给定宽度，且 s 居中
count(s,sub)	子串 sub 在 s 中出现的次数
find(s,sub)	求子串 sub 在 s 中首次出现的位置
join(list)	将列表 list 中的所有字符串合并成一个字符串
ljust(s,width)	将 s 扩展到给定宽度，且 s 居左（左对齐）
lower(s)	将 s 的所有字母改成小写
lstrip(s)	将 s 的所有前导空格删去
replace(s,sub,newsub)	将 s 中所有子串 sub 替换成 newsub
rfind(s,sub)	求子串 sub 在 s 中最后一次出现的位置
rjust(s,width)	将 s 扩展到给定宽度，且 s 居右（右对齐）
rstrip(s)	将 s 的所有尾部空格删去
split(s)	将 s 拆分成子串的列表
upper(s)	将 s 的所有字母改成大写

表 2.5 string 库中的一些函数

下面是几个简单的例子：

```
>>> from string import *
>>> capwords("hello world!")
'Hello World!'
>>> count("知之为知之不知为不知","不知")
2
>>> find("知之为知之不知为不知","不知")
10
>>> rfind("知之为知之不知为不知","不知")
16
>>> print replace("知之为知之不知为不知","知","zhi")
zhi 之为 zhi 之不 zhi 为不 zhi
```

2.4 布尔类型 bool

布尔是 19 世纪英国数学家，他建立了命题代数，简单说就是将逻辑推理变成了代数计 算。所谓命题就是可以判断真假的语句，因此在编程语言中，将真、假两个值构成了一个类 型，即布尔类型，真和假也称为布尔值。以真或假为值的表达式称为布尔表达式，它在程序 设计中的作用是描述某种条件，以支持“如果某条件满足，则执行某语句”之类的处理过程。 第 3 章将学习的条件和循环语句中都会用到布尔表达式。

Python 语言自从 2.3 版之后定义了布尔类型 bool，bool 类型的两个值为 True 和 False。在那之前，Python 分别用 1 和 0 来表示真、假。当然，这个用法一直延续到现在， 详见后文。

2.4.1 关系运算

最简单的布尔表达式是判断两个表达式的值的大小关系的，一般形式是：

```
<表达式> <关系运算符> <表达式>
```

其中两个表达式可以是数值类型或字符串类型的表达式，而关系运算符包括<、<=、>、>=、==、!=（或<>）六种，分别表示小于、小于等于、大于、大于等于、等于和不等于。这些运算符中尤其要注意“等于”运算符，初学者常犯的一个错误是用“=”来表达相等关系，

事实上在 Python 中，“=”是赋值符号，两个等号连写才是“相等”的意思。

数值的大小比较是众所周知的，而字符串的大小比较则不是那么显然。Python 中，字符串是按所谓字典序进行比较的，即基于字母顺序的比较，而字母顺序又是根据 ASCII 编 码顺序确定的。这样，所有大写字母都排在任何小写字母之前，而同为大写字母或同为小写 字母的两个字母之间按字母表顺序排列。至于标点符号、阿拉伯数字等各种字符的顺序也必 须按 ASCII 编码确定大小。例如：

```
>>> 3 &gt; 2
True
>>> 4 + 5 == 5 + 4
True
>>> a = -8
>>> a * 2 &gt; a
False
>>> "like" &lt; "lake"
False
>>> "B-2" &lt; "f-16"
True
>>> 2 = 2
SyntaxError: can't assign to literal
```

2.4.2 逻辑运算

仅用简单布尔表达式是不够的，复杂条件需要用复杂布尔表达式来描述。将多个简单布 尔表达式用逻辑运算符联结起来，即可构成复杂布尔表达式。Python 语言支持的逻辑运算 符有三个：and、or 和 not。

逻辑运算符 and

逻辑运算符 and 联结两个布尔表达式，并得到一个新的布尔表达式。形如：

```
<布尔表达式 1> and <布尔表达式 2>
```

新表达式的值依赖于参加 and 运算的两个布尔表达式的值。具体的依赖关系可以用一个真值表来定义（表 2.6）：

P	*Q*	*P* and *Q*
F	F	F
F	T	F
T	F	F
T	T	T

表 2.6 逻辑运算符 and 的真值表

在表 2.4 中，P 和 Q 表示参加运算的布尔表达式，P and Q 是新的布尔表达式。由于 P 和 Q 各有两种可能的值，所以 P、Q 组合共有四种可能的值组合，每种组合在表中用一行表示。最后一列就是对应于每种组合的 P and Q 的值。从表中可知，P and Q 为真当且仅当 P 为真并且 Q 为真，这也正是 and（并且）的含义。例如：

```
>>> (3 > 2) and (2 > 1)
True
>>> (3 > 2) and (2 > 3)
False
```

顺便说一下，Python 语言允许一种独特的比较表达式形式，该形式在其他编程语言中 是不允许的。请看下例：

```
>>> 3 > 2 > 1
True
>>> 3 >; 2 > 4
False
```

由于这种连续比较的形式在数学中常用，所以初学者很容易接受。但我们不建议读者使 用这种比较形式，因为这种形式毕竟不为绝大多数编程语言所接受。对于复合条件，还是使 用逻辑运算符 and 来表达为好。

逻辑运算符 or

逻辑运算符 or 联结两个布尔表达式，并得到一个新的布尔表达式。形如：

```
<布尔表达式 1> or <布尔表达式 2>
```

新表达式的值依赖于参加 or 运算的两个布尔表达式的值。具体的依赖关系可以用真值表来定义（表 2.7）：

P	*Q*	*P* or *Q*
F	F	F
F	T	T
T	F	T
T	T	T

表 2.7 逻辑运算符 or 的真值表

从表 2.5 可知，P or Q 为假当且仅当 P 为假并且 Q 为假。也就是说，P 和 Q 只要有一个为真，P or Q 就为真，这大体上就是 or（或者）的含义。例如：

```
>>> (3 > 2) or (3 <= 2)
True
>>> (2 > 3) or (2 > 4)
False
```

要注意的是，虽然 or 大体上相当于自然语言中的“或者”，但还是有细微差别的。从 表 2.5 可见，当 P 和 Q 都为真时，P or Q 也为真。而在日常生活中如果说“P 或者 Q”，一 般意味着 P 和 Q 只有一个为真，即有互斥的意义。鱼或熊掌，不可兼得。

逻辑运算符 not

与 and、or 不同，逻辑运算符 not 是对单一布尔表达式进行否定操作，也称为单目运 算符。用法如下：

```
not <布尔表达式>
```

P	not *P*
T	F
F	T

新表达式的值仍可用真值表定义，见表 2.8：

表 2.8 逻辑运算符 not 的真值表 逻辑运算符 not 比较简单，用例如下：

```
>>> not 3 > 2
False
>>> not not 3 > 2
True
```

后面一个语句相当于我们生活中说的双重否定变肯定。 利用三个逻辑运算符可以构造任意复杂的布尔表达式。当复杂布尔表达式中存在多个逻辑运算符的时候，哪个先计算、哪个后计算就成了问题。同算术运算符一样，逻辑运算符也 定义了优先级，复杂表达式的求值依赖于运算符的优先级规则。例如，考虑下列表达式该如 何计算：

```
a or not b and c
```

在 Python 语言中，为逻辑运算符定义的优先级次序是：not > and > or。因此上面的 表达式等价于下面这个加括号的形式：

```
(a or ((not b) and c))
```

其实，与其背诵优先级规则，不如多用括号来明显地指定计算次序。这对程序员来说不但可以减轻记忆负担，更重要的是增强了代码的可读性。 下面看一个例子。设两个乒乓球运动员 A 和 B 打比赛，a 和 b 分别表示两人的得分。

根据规则，一局比赛结束的条件是：A 得到 11 分或者 B 得到 11 分。这个条件可以表示为下列布尔表达式：

```
a == 11 or b == 11
```

当任一运动员得到 11 分，就导致表达式中的一个简单条件为真，根据 or 的定义，整个表 达式也就为真。或者反过来表达，如果还没有满足上述条件，就继续比赛。因此继续比赛的 条件就是：

```
not (a == 11 or b == 11)
```

实际上，乒乓球比赛规则还要复杂一点。当 A 和 B 打到 10 平，规则规定先多得两分者 获胜。将这一特殊情形考虑进去，并结合上面的普通情形，可将结束条件表达为：

```
(a >= 11 and a - b >= 2) or (b >= 11 and b - a >= 2)
```

其含义是：任一方得分达到 11 分以上，并且领先另一人 2 分以上，则一局比赛结束。 这个条件可以稍加简化，即如

```
(a &gt;= 11 or b &gt;= 11) and abs(a - b) &gt;= 2
```

其含义是：当任一方得分达到 11 分以上，并且两人分差超过 2，则一局比赛结束。

2.4.3 布尔代数运算定律*

将实际问题所涉及的条件表达成布尔表达式，并且能对布尔表达式进行演算，这是程序员必须具备的重要能力。前面介绍的逻辑运算符用于表达各种复杂条件，下面介绍用于布尔 表达式演算、推导的一些运算定律。

我们不加证明地罗列一些布尔代数中常用的定律如下，其中 a、b、c 代表任意布尔表 达式。为了不与赋值符号=和比较运算符==混淆，我们用<=>来表示左右相等。

（1）a and False <=> False

（2）a and True <=> a

（3）a or False <=> a

（4）a or True <=> True

从以上四条定律可见，and 类似于二进制算术中的乘法运算，or 类似于加法运算，True 类似于 1，False 类似 0。这不是巧合，事实上，布尔代数和二进制代数本质上是一样的。

下面两条定律称为分配律：

（5）a or (b and c) <=> (a or b) and (a or c)

（6）a and (b or c) <=> (a and b) or (a and c)

对否定的否定当然就是肯定，这就是双重否定律：

（7）not(not a) <=> a

下面两条定律称为 De Morgan 定律，用于将 not 深入到被否定表达式的内部。

（8）not(a or b) <=> (not a) and (not b)

（9）not(a and b) <=> (not a) or (not b)

程序设计中布尔代数运算定律可以用来化简复杂的布尔表达式，以便代码更容易理解。

以上面的继续进行一局比赛的条件为例，

not (a == 11 or b == 11)

<=> (not (a == 11) and not (b == 11))

<=> a != 11 and b != 11

原来的继续比赛条件 not (a == 11 or b == 11)可以直接解读为：当“（a 得到 11 分或者 b 得到 11 分）不是事实”。这似乎不太合乎我们的日常表达方式。通过应用 De Morgan 定律，最后化简为等价的 a != 11 and b != 11，这个表达式可解读为“当 a 不是 11 分并且 b 也不是 11

分”，也许更容易理解一些。

上例为我们展示了一条编程经验：将实际应用中涉及的条件翻译成布尔表达式时，如果 很容易表达某种事件的终止条件，却较难表达该事件的继续条件，那么可以先将终止条件写 下来，然后对它用 not 加以否定，就得到了继续条件，最后再利用 De Morgan 定律简化这 个继续条件。

2.4.4 Python 中真假的表示与计算*

如前所述，较新版本的 Python 引入了内建类型 bool，并且定义了布尔值 True 和 False。而在此之前，Python 曾经利用 1 和 0 来作为布尔值。

事实上，如今的 Python 在表达真假方面更加灵活——任何内建类型的值都可以解释成布尔值。例如，数值（int、long、float）可以解释成布尔值：0 为 False，非 0 值为 True。又如，字符串也可以解释成布尔值：空串为 False，非空字符串为 True。以后介 绍的列表、元组等数据类型的值也都可以解释为布尔值。

Python 对布尔值的灵活处理方式也影响了逻辑运算符的含义。在 Python 中，布尔表达 式的结果不仅可以是“正宗的”布尔值 True 和 False，还可以是如上所述的各种“非正 宗”布尔值。下面介绍 Python 在实现逻辑运算符采取的策略，我们用 a、b 表示任何表达式（不一定是布尔表达式！）。

（1）a and b：如果 a 的值可解释为 False，则返回 a 的值；否则返回 b 的值。

（2）a or b：如果 a 的值可解释为 False，则返回 b 的值；否则返回 a 的值。

（3）not a：如果 a 的值可解释为 False，则返回 True；否则返回 False。

这些规则看上去有点怪，但仔细思考之后就能理解，它们并没有违反基本的逻辑运算的定义。

以 a and b 为例分析如下：当 a 的值不能解释为 True，我们就不必计算 b 的值，而是直 接返回 a 的值（可解释为 False）作为整个表达式的值；当 a 的值可解释为 True，那么 整个表达式的真假就取决于 b 的值，b 真则表达式真，b 假则表达式假，因此我们可以返回 b 的值作为表达式的值。总之，当且仅当 a 和 b 都可解释为 True 时，表达式 a and b 的 值才可解释为 True。这是完全符合逻辑运算定义的。

对于 or 和 not 也是一样。 下面看几个例子：

```
>>> 2 and "hello"
'hello'
```

说明：Python 先计算 2，发现它是非 0 值，可解释为 True，于是按上述规则（1），返回"hello"的值作为结果。

```
>>> (4 &gt; 5) or 3
3
```

说明：Python 先计算(4 > 5)，发现结果为 False，根据上述规则（2），返回 3 的值作为 表达式的值。

```
>>> s = "hello"
>>> not s[3:3]
True
```

说明：Python 先计算 s[3:3]的值，即字符串 s 的切分操作，但这个索引区间是不成立的： 不可能从索引 3 开始，又以 3 作为结束的上限（因为索引区间的上限是不包含在内的）。因 此切分结果是空串。空串被解释为 False，根据上述规则（3），返回 True。

综上所述，Python 对布尔值和布尔运算的处理很灵活，有时能够便利程序设计。但代价是布尔表达式变得难以理解，很容易导致微妙的错误，所以建议读者慎用。

2.5 列表和元组类型

整数类型、浮点数类型和布尔类型都是最简单的“原子”数据类型，因为这些类型的值

是不可分割的基本数据项。而字符串类型稍微有点复杂，因为字符串可以看成是由许多单字符组成的有序的集合体，我们可以通过索引操作来深入到字符串内部访问其成员。不过，通常我们仍然将字符串类型归为简单的基本类型，毕竟构成字符串的成员是非常简单的单字符。

对于单个数据，我们可以用一个变量来存储。假如程序需要处理 10 个数据，那么我们可以在程序中定义 10 个变量来存储。但是，如果程序中需要处理成千上万个数据，怎么办？如果定义成千上万个变量来分别存储数据，显然是很不方便的，而且非常容易出错。这时，我们就希望能用一个变量来存储大量数据的集合体。事实上，Python 语言提供了多种集合体类型，包括列表、元组、字典和文件。这些类型的值都不是原子值，而是由很多值聚在一起构成的复合值。

本节先简要介绍列表和元组类型，关于集合体类型更多更详细的内容将在第 6 章介绍。

2.5.1 列表类型 list

列表（list）是由若干数据组成的序列（sequence）①。构成列表的数据既能作为一个整 体去参加运算，也可以作为个体去参加运算。现实世界中列表是很常见的数据，如名单、待 办事项清单、数学中的数列等都可表示为列表。Python 提供了内建类型 list 以支持列表数 据的表示和操作。

列表的表示

Python 列表类型的字面值采用如下形式：

```
[<表达式 1>, <表达式 2>, ..., <表达式 n>]
```

即用一对方括号将以逗号分隔的若干数据（表达式的值）括起来。

列表中成员的个数称为列表的长度，可以用 len()函数求得。 就像数学里有空集一样，不含任何成员的列表也是有意义的，称为空列表，用一对方括号[]表示。空列表的长度当然为 0。 可以将列表字面值赋给变量，以便将来通过变量引用该列表。 下面的语句演示了列表的类型、字面值、长度等基本概念：

```
>>> type([1,3,5,7,9])
&lt;type 'list'&gt;
>>> len([1,3,5,7,9])
5
>>> ["list","sequence"]
['list', 'sequence']
>>> print [],len([])
[] 0
>>> x = ['apple','banana','orange']
>>> type(x)
<type 'list'>
```

① 列表和序列几乎是同义词，但本书对两个术语的用法做了区分。序列用作更一般的术语，列表只是序列 的特例。例如，和列表一样，字符串、元组也可视为序列的特例。

```
>>> print x
['apple', 'banana', 'orange']
```

很多编程语言都提供一种称为数组（array）的数据类型，数组可以说是列表的特例。 数组的特殊之处有两点：一是固定长度，即成员个数是固定的；二是各成员是同类型的。因 此我们常说程序中定义了一个“长度为 10 的整数数组”或者“长度为 5 的字符串数组”等 等。而 Python 的列表类型没有这两条限制，不但列表长度可以动态改变，而且列表的成员 可以是不同类型的数据。例如，下面这个列表由整数、浮点数、字符串和布尔值四种类型的 数据构成：

```
>>> y = [123,"apple",3.14,True]
>>> y
[123, 'apple', 3.14, True]
```

列表的成员本身也可以是列表，如：

```
>>> z = ["my favorite",["apple","pear"],3.14,[True,False]]
>>> print z
['my favorite', ['apple', 'pear'], 3.14, [True, False]]
```

计算机应用于数学计算时，经常需要表示数学中的矩阵，显然矩阵可以用以列表为成员 的列表很轻松地表示出来。例如下面的列表 m 就表示了一个 2×3 阶的矩阵：

```
>>> m = [[11,12,13],[21,22,23]]
>>> print m
[[11, 12, 13], [21, 22, 23]]
```

列表的操作

为了对列表进行操作，Python 提供了列表成员的索引机制，即通过位置编号来引用列 表成员。列表中第一个成员的索引为 0，第二个成员的索引为 1，其余依此类推。也可以从 后往前编号：最后一个成员的索引是-1，倒数第二个成员的索引是-2，其余依此类推。通 过索引操作访问列表成员的一般形式如下：

<列表>[<数值表达式>] 其中数值表达式的值就是位置索引，整个索引操作的返回结果就是索引位置上的成员。如果 索引超出了范围，则导致出错。

接着前面的例子，我们来通过索引访问列表成员 ：

```
>>> x[0]
'apple'
>>> x[-1]
'orange'
>>> i = 0
>>> x[i+1]
'banana'
>>> x[3]
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module> x[3]
IndexError: list index out of range
>>> print y[3],y[1]
True apple
>>> m[0]
[11, 12, 13]
>>> m[0][1]
12
```

其中最后两个例子显示，我们可以用 m[0]来访问矩阵 m 的第一行，用 m[0][1]来访问矩 阵 m 的第一行、第二列的元素。

Python 也支持通过指定列表的一个索引区间来访问列表的“子列表”，一般形式是：

```
<列表>[开始位置:结束位置]
```

其中开始位置和结束位置都是 int 类型的表达式，整个操作的含义是返回从开始位置到结束位置（不含）的子列表。开始位置和结束位置是可选的，在未指定的情况下，Python 默认开始位置为 0，结束位置为 n（列表长度）。仍然延续上面的例子：

```
>>> x[0:2]
['apple', 'banana']
>>> x[1:]
['banana', 'pear']
>>> x[:-1]
['apple', 'banana']
```

我们看到，列表的索引机制和前面学过的字符串类型很像。这一点都不奇怪，因为字符 串可以看作是列表的特例——由字符组成的列表。对字符串能执行的操作，对列表也是可以 的。因此，前面学过的字符串运算+和*，也适用于列表，可以实现列表的合并、复制操作。 例如：

```
>>> [1,3,5] + [2,4]
[1, 3, 5, 2, 4]
>>> 10 * [0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

然而，列表和字符串有一个重大不同：字符串是不可更改的，而列表是可以更改的。我 们可以为列表增加成员、删除成员、改变某个成员的值等等。延续前面的例子演示如下：

```
>>> x[2] = "pear"
>>> x
['apple', 'banana', 'pear']
>>> x = x + ["peach"]
>>> x
['apple', 'banana', 'pear', 'peach']
>>> del x[1]
>>> x
['apple', 'pear', 'peach']
```

以上语句首先将列表 x 的第 3 个成员从'orange'改成了'pear'，然后为 x 增加了第 4 个 成员'peach'，最后将 x 的第 2 个成员'banana'删除。这里 del()是 Python 的内建函数， 用于删除数据。

注意，增加、修改、删除操作除了可以像以上例子一样针对单个列表成员进行，也可以针对列表的一个片段进行。

Python 还支持对列表的许多其他操作，包括搜索列表以查找特定数据、在列表中间插 入数据、给列表排序等等，将在第 6 章中介绍。

range()函数

Python 语言提供了一个内建函数 range()，用于产生整数列表。我们在第 1 章中已经 见到它的用法，这里给出其完整的用法介绍。

range()的一般形式是： range(<起点>, <终点>, <步长>)

返回结果是从起点到终点的有序整数列表，各整数之间以步长为差。要特别注意一点，终点的含义是说列表中的整数不得超过终点，但它本身是不包含在列表当中的，对此初学者很容易犯错。另外，起点或步长是可以省略的，它们的缺省值分别是 0 和 1。因此，range 函数的使用方式有以下三种：

```
range(n)：产生整数列表[0,1,2,...,n-1]
range(i,j)：产生整数列表[i,i+1,i+2, ..., j-1]
range(i,j,s)：产生整数列表[i,i+s,i+2s,...]
```

其中第三种形式的返回结果取决于步长 s，不一定以 j-1 作为最后一个成员。

下面看几个例子：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> range(10,0,-3)
[10, 7, 4, 1]
>>> range(1,1)
[]
```

从例中可见，当步长为正数时产生递增的列表，当步长为负数时产生递减的列表。最后一个例子表明，如果没有满足条件的整数（从 1 开始并且小于 1 的整数是不存在的），则产生空列表。

range()函数常和 for 循环语句连用，详见第 3 章。

2.5.2 元组类型 tuple

和列表类似，元组也是数据集合体的一种。尽管很多编程语言都没有提供内建的元组数 据类型，但实际上元组类型是非常有用的。在数学中，表示平面或空间中的点需要用到元组 (x,y) 或(x,y,z)，一般的向量也是元组 v = (v1, ..., vn)。现实中很多信息都可以表示为元组，例 如一对夫妻可以表示为形如(husband,wife)的二元组，超市购物打印出来的单据是形如(商品 名称,单价,数量,总价)的元组的列表，通讯录中记录了大量形如(姓名,电话,地址)的元组，等 等。

Python 提供了元组类型 tuple，该类型的字面值形式是用一对圆括号括起来并以逗号 分隔的多个成员。例如：

```
>>> t = (1,2,3)
>>> t
(1, 2, 3)
>>> type(t)
<type 'tuple'>
```

和空列表一样，没有成员的元组是空元组，用()表示。比较特殊的是，如果元组只有 一个成员，仍然需要在该成员后面加上逗号，例如：

```
>>> (8,)
(8,)
>>> (8)
8
```

可见 Python 将(8)解释为单个数值 8，而不是元组。 和列表一样，可以通过索引来访问元组的成员。例如：

```
>>> t[0] 1
>>> t[0:2]
(1, 2)
```

注意，元组值用圆括号，通过索引访问元组的成员则用方括号。 同样地，列表运算基本上都适用于元组。例如：

```
>>> t + (4,5) (1, 2, 3, 4, 5)
>>> 3 * t
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> len(t) 3
```

但是，元组和列表之间有个重要的不同：元组是不可更改的。一旦创建了元组，该元组就不能修改、添加、 删除成员。在这一点上元组和字符串是相似的。例如如果要将元组 t 的 第 3 个分量改为 8，下面的做法是不行的：

```
>>> t[2] = 8
Traceback (most recent call last):
File "<pyshell#43>", line 1, in <module> t[2] = 8
TypeError: 'tuple' object does not support item assignment
```

实在想修改元组的话，只能通过创建新的元组来迂回达到目的。例如：

```
>>> t = t[0:2] + (8,)
>>> t (1, 2, 8)
```

例中将 t 的前两个成员和单元素元组(8,)合并，创建了一个新元组，然后将此元组赋值给t。

更多关于元组的知识将在第 6 章介绍。

2.6 数据的输入和输出

任何程序都需要与用户进行沟通，这就要求程序具有输入输出的功能。输入是指程序从

用户那儿获取数据，输出是指程序向用户显示或打印数据。 程序中负责与用户沟通的部分称为用户界面，它是程序设计的一个重要组成部分。设计

用户界面时要遵循的一个主要原则是所谓“用户友好性”，即要让用户在与计算机程序交互 时感到非常简单、方便和不易犯错。本章只讨论简单的数据输入输出，本书后文将专门讨论 复杂的图形用户界面的程序设计问题。

2.6.1 数据的输入

有的程序处理的是静态数据，即在程序运行之前数据已准备好。这时我们可以预先将数据存储在变量之中，并且能够针对数据的特性来选用合适的处理命令。例如，已知 Lucy 在 2012 年是 7 岁，则可编写下面的程序来显示 Lucy 的出生年份信息：

【程序 2.1】eg2_1.py

```
name = "Lucy"
age = 7
birthYear = 2012 - age
```

print name,"was born in",str(birthYear)+"." 程序中，预定的数据分别存储在变量 name 和 age 中，利用算术表达式 2012-age 求得出 生年份，利用 str 函数将年份转换成字符串类型，利用字符串合并运算+为输出信息添上句 点。运行此程序，无需用户参与即可直接得到下面的结果：

```
Lucy was born in 2005.
```

而另一些程序要处理的数据则是在执行程序时由用户提供的。用户提供数据的方式有多 种，其中最简单的方式是在程序中使用输入语句，其他方式包括在启动程序时以命令行参数 的方式传递数据或在图形用户界面中利用输入构件来提供数据。在此我们讨论最简单的输入 语句方式。

Python 中提供了 input()函数用于输入数据，该函数通常的使用方式如下：

```
<变量名> = input(<提示字符串>)
```

执行时首先在屏幕上显示提示字符串，然后等待用户输入（以回车键表示输入完毕），并将用户输 入作为一个表达式进行解释、求值，最后将求值结果赋予变量。例如：

```
>>> x = input("请输入: ")
请输入: 123
>>> x
123
>>> x = input("请输入: ")
请输入: 1+2
>>> x
3
```

可见，当用户连续按下数字键 1、2、3、回车键之后，input 函数将 123 视为表达式进行求值，结果即数值 123。而当用户按下数字键 1、加号键+、数字键 2、回车键之后，input 将 1+2 视为表达式进行求值，结果为数值 3。

当然，作为一个函数，input 也可以直接用在表达式中，其作用相当于一个值。例如：

```
>>> 3 + input("请输入：")
请输入：4
7
```

input 不仅能接收数值类型的表达式，也能接收其他类型的表达式。例如：

```
>>> x = input("请输入: ")
请输入: "123"
>>> x
'123'
>>> x = input("请输入: ")
请输入: "1"+"2"
>>> x
'12'
>>> x = input("请输入: ")
请输入: True and False
>>> x
False
```

可见，当用户连续按下引号键"、数字键 1、2、3、引号键"、回车键之后，input 将"123" 视为表达式进行求值，得到的结果即为字符串"123"。而当用户连续按下引号键"、数字键 1、引号键"、加号键+、引号键"、数字键 2、引号键"、回车键之后，input 将"1"+"2" 视为字符串运算表达式进行求值，得到结果"12"。第三个输入例子是布尔表达式，结果是 显然的。

下面我们将程序 2.1 改写成另一版本：由用户输入姓名和年龄，然后计算出生年份。

【程序 2.】eg2_2.py

```
name = input("Name: ")
age = input("Age: ")
birthYear = 2012 - age
print name,"was born in",str(birthYear)+"."
```

以下是程序 2.2 的一次执行示例：

```
>>> import eg2_2
Name: "Lucy" Age: 7
Lucy was born in 2005.
```

从上面的例子可以看到，input 函数在输入数值型数据时很方便，但在接收字符串类 型的数据时有点麻烦，因为要为字符串数据加上引号。如果不加引号，input 会将输入的 字符串解释为变量名，以便构成合法的表达式。除非程序中定义过该变量，否则会导致"变 量未定义"的错误。例如：

```
>>> x = input("请输入：")
请输入：Lucy
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module> x=input("请输入：")
File "<string>", line 1, in <module> NameError: name 'Lucy' is not defined
>>> Lucy = 7
>>> x = input("请输入：")
请输入：Lucy
>>> x
7
```

其实，Python 还提供了另一个输入函数 raw_input()，它用于字符串数据输入时更方 便。raw_input 函数通常的使用方式如下：

<变量名> = raw_input(<提示字符串>) 执行时首先在屏幕上显示提示字符串，然后等待用户输入（以回车键表示输入完毕），用户 键入的所有内容视为一个普通的字符串而不是表达式，该字符串就是 raw_input 的返回 值，可以赋值给其他变量。例如：

```
>>> x = raw_input("请输入：")
请输入：hello world
>>> x
'hello world'
```

可见，raw_input 将用户键入的所有字符构成一个字符串并作为函数的返回值。因此，用 raw_input 输入字符串时不需要加引号，比 input 略为方便些。

同样可以将 raw_input 函数直接用在某个表达式中，其作用相当于一个字符串。例如：

```
>>> 2 * raw_input("请输入：")
请输入：Hello
'HelloHello'
```

input 与 raw_input 的比较 根据上面的介绍可知，如果需要输入数值或数值表达式，最好用 input；如果需要输入字符串，最好使用 raw_input。但这不是绝对的，实际应用中经常也用 raw_input 输入数值数据，具体做法是：先作为字符串输入，然后通过类型转换函数（int、long、float）或 eval 函数来将字符串转换成数值。例如：

```
>>> x = int(raw_input("Please enter a number: "))
Please enter a number: 123
>>> x + 456
579
```

例中 raw_input 所接收的输入字符串被 int 函数转换成整数类型。这看起来比直接使用 input 来输入数值麻烦，但 raw_input 有个好处是能处理空输入的情况（即用户直接按 回车键），而使用 input 时空输入会导致错误。试比较：

```
>>> x = input("Press Enter: ")
Press Enter:
Traceback (most recent call last):
File "&lt;pyshell#15&gt;", line 1, in &lt;module&gt; x = input("Press Enter: ")
File "&lt;string&gt;", line 0
^
SyntaxError: unexpected EOF while parsing
>>> x = raw_input("Press Enter: ")
Press Enter:
>>> x
''
```

2.6.2 数据的输出

Python 中最简单的输出方式是使用我们早已见过的 print 语句。print 语句用于在屏

幕上显示信息，其用法可以概括为以下几种模板①：

```
print
print <表达式>
print <表达式 1>, <表达式 2>, ... , <表达式 n>
print <表达式 1>, <表达式 2>, ... , <表达式 n>,
```

print 语句的语法可简述为：print 后面可以出现零个、一个乃至 n 个表达式，各表达式之间用逗号分隔。print 语句的语义是：从左到右计算每一个表达式，将各表达式的值以文本形式从左到右显示在屏幕的同一行上，值与值之间插入一个空格作为间隔。没有表达式的 print 语句（见第一个模板）用于输出一个空白行。通常情况下，连续两条 print 语句 将在屏幕的两个不同行上显示信息，但如果前一条 print 语句以逗号结尾（见第四个模板）， 则下一条 print 语句将不会换行，而是接在前一行的后面继续显示。例如：

【程序 2.3】eg2_3.py

```
x = 1+2*3/4
print "1+2*3/4 =", x print
print "蜀道难",
print "难于上青天"
print "蜀道难" + "难于上青天"
```

下面是程序 2.3 的一个执行示例：

```
>>> import eg2_3
1+2*3/4 = 2
蜀道难 难于上青天 蜀道难难于上青天
```

注意上面输出中最后两行的细微区别：将两个数据在同一行上输出，不同的做法会导致两个数据之间是否有空格的差别。

2.6.3 格式化输出

很多应用都要求将数据按整齐的格式输出，用 print 语句能够安排简单的格式。例如， 下面的程序画出一棵简单的圣诞树：

【程序 2.4】eg2_4.py

```
print "    *    "
print "   **@   "
print "  *@***  "
print " *****@* "
print "*********"
print "    *    "
print "    *    "
```

可见，数据的输出位置、间隔空白等都是用最直接的手动方式安排的。执行结果如图 2.3 所示。

> ① Python 3.x 与 Python 2.x 的一个重要区别是将 print 实现为一个函数，即 print(<表达式>)。

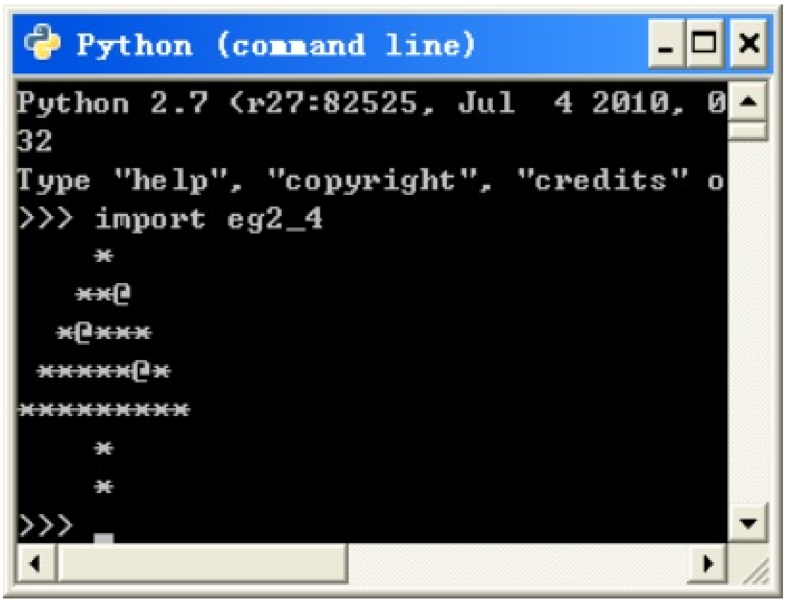


图 2.3 程序 2.4 的执行结果

如果需要设计复杂的输出格式，仅靠 print 就无能为力了，事实上 Python 提供了更好 的做法——字符串格式化运算。先看个简单例子：在财会、银行应用中，输出金额数据时有 习惯的固定格式，如一元伍角不应显示成 1.5，而应显示成￥1.50。而用普通的 print 语 句无法保留末尾的 0：

```
>>> print 1.50
1.5
```

为了解决这个问题，我们可以采用 Python 的格式化输出的功能：

```
>>> x = 1.5
>>> print "The amount is ￥%0.2f" % (x)
The amount is ￥1.50
```

这里用到了 Python 的字符串格式化运算符%。

字符串格式化运算%的一般用法如下：

```
<模板字符串> % (<数据 1>, ..., <数据 n>)
```

这个运算的结果是一个字符串，该字符串是按照模板字符串的样子构造的。模板字符串中一般会留有一些“空位”，需要用实际数据填入这些空位。显然，空位的个数和实际数据的个 数必须对应一致。总之，向模板字符串的空位中填入了实际数据后，所得字符串就是格式化 运算的结果。

每个“空位”实际上是一个格式定义符，用于描述对填入数据的格式化处理。如上面的 例子中，模板字符串"The amount is ￥%0.2f"包含一个空位，即格式定义符%0.2f。 数据 x 的值将按照所定义的格式填充到这个空位中。具体是怎样的格式呢？

格式定义符的一般形式是：

```
%<宽度>.<精度><类型字符>
```

模板字符串中出现的格式定义符以%开头，表示一个空位，注意不要与模板字符串后面的格式化运算符%混淆。格式定义符以类型字符结尾，表示向该空位填入的数据将格式化为特定数据类型，常用的类型字符有十进制整数 d、浮点数 f、字符串 s 等。

格式定义符的中间部分包括宽度、小数点和精度，其中宽度表示“空位”的预留空间宽度（以字符计），精度用于浮点数格式，表示小数点后保留几位数字。如果实际数据的宽度 超出空位的预留宽度，则空位自动扩张至所需宽度；如果实际数据的宽度小于预留宽度，则 按预留宽度输出（这时会多出一些空白）。若省略宽度或指定宽度为 0，则表示根据实际数 据的宽度分配空间。

汇总以上说明，即可明白格式定义符%0.2f 的意思是：数据按浮点数类型格式化，根 据数据的实际宽度分配空间，保留两位小数。

另外补充两点：第一，当预留宽度大于数据的实际宽度时，该数据在预留空间内默认是 右对齐的，但可以通过在宽度前加上负号改成左对齐，如%-8.2f。第二，浮点数输出时默 认的精度是保留 6 位小数，实际数据的小数部分不足 6 位时自动补 0，超出 6 位时自动进行 舍入；如果指定过高的精度，可能导致无法精确表示。

建议读者试用各种格式定义符，以便熟练掌握格式化输出的功能。下面是一些例子：

```
>>> "Formatted as an int: %d" % (3.14159265)
'Formatted as an int: 3'
>>> "Formatted as a float: %f" % (3.14159265)
'Formatted as a float: 3.141593'
>>> "Formatted as a float: %.4f" % (3.14159265)
'Formatted as a float: 3.1416'
>>> "Formatted as a float %.20f" % (3.14159265)
'Formatted as a float 3.14159265000000020862'
>>> "Formatted as a string: %s" % (3.14159265)
'Formatted as a string: 3.14159265'
>>> "Formatted as a string: %20s" % (3.14159265)
'Formatted as a string: 3.14159265'
>>> "Formatted as a string: %-20s" % (3.14159265)
'Formatted as a string: 3.14159265 '
>>> "%s has won $%d!" % ("Mr. Smith",10000)
'Mr. Smith has won $10000!'
>>> "%s gives %s $%d." % ("Tom","Tim",100)
'Tom gives Tim $100.'
```

2.7 编程案例：查找问题

下面我们通过一个简单程序来综合应用本章所介绍的知识。

实际应用中经常遇到“查找”问题：即从一个数据集中查找我们需要的数据。查找技术 是程序设计的一个重要技术，存在着许多高效的查找算法。在此，我们考虑一种很简单的查 找问题。场景：下面我们编一个小程序。基本的 IPO 模式。

假如我们要编一个程序，它接收用户输入的月份数值（1～12），并输出对应月份的英文 缩写。例如，当用户输入 3，则程序输出 Mar。虽然我们还没有学习 Python 的控制流语句（见下一章），但我们可以利用字符串操作来完成程序功能。

我们首先将所有月份的英文缩写保存在一个字符串之中：

```
months = "JanFebMarAprMayJunJulAugSepOctNovDec"
```

此处的 months 相当于数据集，接着我们需要根据用户输入的月份数值从这个数据集中查 找相应的缩写（子串）。如何根据用户输入的 m 找到相应子串呢？

程序设计往往需要为应用问题建立数学模型。本查找问题的模型是很简单的，由于数据 集中每个月份名称缩写长度都是 3，因此只要找到相应月份的开始位置 pos，再截取长度为 3 的子串即可：

```
monthAbbr = months[pos:pos+3]
```

于是问题演变成了如何根据用户输入的月份值 m 找到开始位置 pos。试着确定几个月 份的索引开始位置：

```
m = 1 pos = 0
m = 2 pos = 3
m = 3 pos = 6
```

由此不难推知 m 月的索引开始位置是(m-1)*3。 通过以上分析，我们设计出程序的算法：

```
输入月份值 m；
计算在数据集中的索引开始位置(m-1)*3 并取子串；
输出月份名称缩写。
```

这是最简单的 IPO 算法模式，即“输入－处理－输出”的模式。下面我们来实现这个 算法。

【程序 2.5】eg2_5.py

```
months = "JanFebMarAprMayJunJulAugSepOctNovDec" m = input("Enter a month number (1-12): "
pos = (m-1) * 3
monthAbbr = months[pos:pos+3]
print "The month abbreviation is", monthAbbr + "."
```

下面是程序 2.5 的运行实例：

```
>>> import eg2_5
Enter a month number (1-12): 3
The month abbreviation is Mar.
```

2.8 练习

1. 什么是数据？什么是数据类型？

2. Python 中的数值类型有哪些？对数值类型能执行什么运算？

3. Python 中的字符串有哪些表示方式？对字符串类型能执行什么运算？

4. Python 中的布尔类型提供了哪两个值？对布尔类型数据能执行什么运算？

5. Python 中的列表类型和其他编程语言中常见的数组类型有何异同？

6. Python 中的字符串类型和列表类型有何异同？

7. Python 中的元组类型与列表类型有何异同？

8. 说 Python 中的变量是动态类型的，这是什么意思？

9. 哪些运算符针对不同类型的数据有不同意义？

10. 利用 Python 计算以下表达式。如果出错，找出原因。

（1） `4.0 / 10.0 + 3.5 * 2`

（2） `10 % 4 + 6 /2`

（3） `abs(4 ? 20 / 3) ** 3`

（4） `sqrt(4.5 ? 5.0) + 7 * 3`

（5） `3 * 10 / 3 + 10 % 3`

（6） `3L ** 3`

11. 将下列数学式用 Python 表达式表示出来。假设已通过 import math 导入了数学库。

（1） `(a + b) ´ c`

（2） `n(n - 1) / 2`

（3） $2\pi r$

（4） $\sqrt{r(\cos a)^2 + r(\sin a)^2}$

（5） `(y2 - y1)/(x2 - x1)`

12. 设计程序：输入球体半径 r，计算球体的体积（$4/3\pi r^3$）和表面积（$4\pi r^2$）。

13. 设计程序：输入平面上两个点的坐标(x1,y1)和(x2,y2)，计算两点间距离。距离公式为 d = $\sqrt{(x2-x1)^2+(y2-y1)^2}$ 。

14. 设计程序：输入以英尺英寸为单位的身高数据，转换成以米为单位的数据。（1 英尺＝12 英寸＝0.305 米）

15. 设计程序：输入 5 个考试分数，计算平均分。

16. 假设已经执行了如下语句：

```
>>> import string
>>> s1 = "programming"
>>> s2 = "language"
```

（1）求下列各表达式的值。

a） `s1[1]`

b） `s1[:4]`

c） `s1[0] + s2[:3]`

d） `s2[5:len(s2)]`

e） `"Python " + s2`

f） `2 * (s1[:2] + s2[-1])`

g） `string.count(s1,'r') + string.find(s1,'r')`

h） `string.ljust(string.upper(s2),10)`

（2）利用 s1、s2 和字符串操作，写出能产生下列结果的表达式。

a） `'gram'`

b） `'ProgLang'`

c） `'la la la'`

d） `' language '`

e） `'progrAmming lAnguAge'`

17. 求下列字符串格式化操作的结果。如果出错，解释原因。

（1） `"%s has won %d gold medals." % ("China",38)`

（2） `"Hello %s." % ("Tom","Tim")`

（3） `"%0.2f %0.2f" % (3.1416,2.718)`

（4） `"Time left %02d:%05.2f" % (1,5.432)`

（5） `"%3d" % ("14")`

18. 将下列条件用布尔表达式表示出来：

（1）a 大于 b，或者 a 小于等于 b 且 c 不等于 0。

（2）a 和 b 的差不超过 0.005。

（3）字符串 s 以“水”开头，并且包含子串“酒旗”。

（4）字符串 s1 的长度为 10，或者 s1 在字符串 s2 中出现 2 次。

19. 若 P 表示“x 小于等于 y 并且 x 大于 0”，那么 not P 表示什么？

20. 假设已经执行了如下语句：

```
>>> s1 = [1,2,3,4]
>>> s2 = ['a','b','c']
```

求下列各表达式的值。

（1） `s1 + s2`

（2） `2 * s1 + 3 * s2`

（3） `s1[:3]`

（4） `s2[0:len(s2)]`

21. 将第 20 题中的 s1 和 s2 改为元组，重做（1）～（4）。

22. 设计程序：输入五分制的分数（0～5），输出相应的等级制分数：5＝A，4＝B，3＝C，2＝D，1＝F，0＝F。（提示：仿照程序 2.5 的查找方法）

23. 设计程序：输入百分制的分数（0～100），输出相应的等级制分数：90～100＝A，80～89＝B，70～79＝C，60～69＝D，0～59＝F。

第 3 章 数据处理的流程控制

计算机程序是对特定数据进行特定操作的一系列编排好的处理步骤。第 2 章介绍了各种 类型的数据的表示和操作，本章介绍如何“编排”处理步骤的问题，即程序的流程控制。编 程语言①提供了控制流语句，用于控制程序从多条执行路径中选择一条路径执行下去。不同 语言支持的控制流语句在形式上可能各不相同，但其作用是相同的，大致可分为顺序、无条 件跳转、条件分支、循环、子程序等几类控制结构。

结构化程序设计方法的基本思想是只用顺序、条件分支和循环三种控制结构来编制程序， 并使整个程序由具有唯一入口和唯一出口的语句块相互串联、嵌套而成。这样的程序具有结 构清晰、易理解、易验证和易维护等优点。

3.1 顺序控制结构

程序是一个语句序列，执行程序就是按特定的次序执行程序中的语句。程序中执行点的

变迁称为控制流程，当执行到程序中的某一条语句时，也说控制转到了该语句。由于复杂问题的解法可能涉及复杂的执行次序，因此编程语言必须提供表达复杂控制流程的手段，称为编程语言的控制结构。

程序的控制流程可以用流程图（flowchart）来形象地表示。流程图采用标准化的图形符 号来描述程序的执行步骤，是一种常用的程序设计工具。在较低的抽象级上，流程图中的每 一个步骤可能都是单条语句，而在较高的抽象级上，每个步骤都可以是由多条语句构成的语 句块。本书中不另辟章节来系统地介绍各种标准的流程控制符号，而是通过例子演示常用流 程控制图形符号及其用法，因为这些内容是非常直观易懂的。

最简单的控制结构是顺序控制结构。编程语言并不提供专门的控制流语句来表达顺序控 制结构，而是用程序语句的自然排列顺序来表达。计算机按此顺序逐条执行语句，当一条语 句执行完毕，控制自动转到下一条语句。

现实世界中这种顺序处理的情况是非常普遍的，例如我们接受学校教育一般都是先上小 学，再上中学，再上大学；又如我们烧菜一般都是先热油锅，再将蔬菜入锅翻炒，再加盐加 佐料，最后装盘。如果一个处理过程由顺序执行的步骤 S1、S2、…、Sn 组成，用流程图表示的话即如图 3.1 所示：

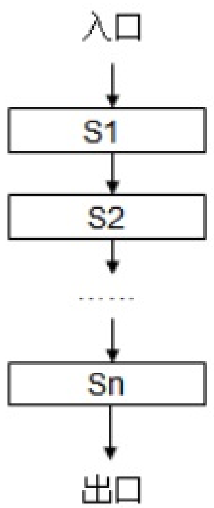


图 3.1 顺序控制结构

> ① 指命令式（或过程式）编程语言。函数式和逻辑式编程语言中没有这里所说的控制流语句。

作为例子，我们来写一个顺序控制结构的简单程序——温度转换程序。当中国人去美国旅游，听到导游说当地气温是 80 度，一定会感到困惑。其实美国人用的是华氏温标，与中国人用的摄氏温标不同。如果能写一个程序将华氏温度转换成摄氏温度，就可以帮助中国游客 知

冷知热。实现温度转换的算法非常简单，只需顺序执行三个步骤：输入华氏温度值；转换 成摄氏温度值；输出摄氏温度值。下面是这个算法的流程图（图 3.2）及 Python 实现：

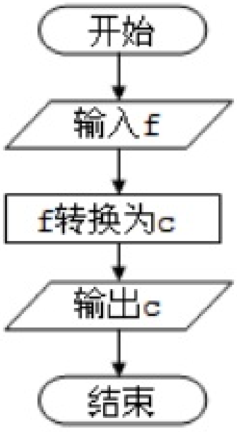


图 3.2 温度转换算法

【程序 3.1】eg3_1.py

```
f = input("Temperature in degrees Farenheit: ")
c = (f – 32) * 5.0 / 9
print "Temperature in degrees Celsius:", c
```

执行这个程序，并输入 80，将看到屏幕显示转换结果是摄氏 26.6666666667 度，是一个 适合旅游的舒适温度。

图 3.2 中的三个步骤（除了开始、结束）恰好可以用程序 3.1 中的三条语句实现，但如前 所述，我们可以在比语句更高的级别上来考虑顺序执行的步骤。图 3.1 中的诸 Si 不一定对应 着单条语句，完全可以是一个语句块，并且这个语句块本身可由各种控制结构组成。例如程 序 3.1 的三个步骤就可以构成别的程序的一个步骤，如图 3.3 所示：

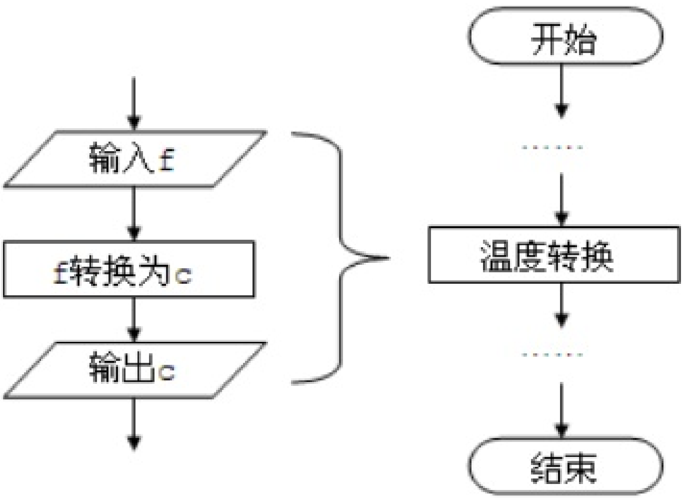


图 3.3 低级别步骤抽象成高级别步骤

这种将若干低级别步骤看成整体并构成一个高级别步骤的做法也是抽象的一种形式，是程序设计中广泛使用的思维方式，对此在 3.5.2 中有更一般的阐述。顺序控制结构是最简单、最普遍的控制结构，计算机执行程序时的缺省控制流就是语句的自然排列顺序。但是，仅靠顺序执行的步骤是不足以解决复杂问题的，复杂问题一般需要 根据情况来改变执行顺序。

3.2 分支控制结构

我们都有这样的生活经验："道路"——不管它指的是具体道路，还是指"人生道路"这样的抽象道路——一般都不是能够笔直一条路走到底的，我们会时不时遇到岔路口，需要根 据一些条件来决定选择哪一条路继续前行。程序的控制流程也是一样，一般都不是从第一条 语句一直顺序执行到最后一条语句，而是在执行过程中需要根据不同情况来选择执行不同的 语句序列。编程语言中提供了根据条件来选择执行路径的控制结构，称为分支控制结构，也 称为条件或判断结构。

3.2.1 单分支结构

下面我们来改进程序 3.1，使得程序能向游客提供一些温馨提示，例如当温度达到摄氏 35 度就发出高温警告信息。显然这里需要判断温度是否高于 35 度，并根据是或否来执行不 同的动作。

所有编程语言都提供了条件语句（if 语句），用来实现有条件地执行语句的功能。Python 语言的 if 语句有多种形式，最简单的形式是：

```
if <条件表达式>:
    <条件语句体>
```

其中<条件表达式>是布尔表达式，<条件语句体>是由一条或多条语句组成的语句序列。<条件语句体>的左端与 if 部分相比必须向右缩进，表明它是 if 部分（不妨理解为条件语句的头部）的下属，就像躯体是头部的下属一样。

if 语句的语义很容易理解：首先计算 if 后面的条件表达式，如果结果为 True，则控制转 到条件语句体的第一条语句，一旦条件语句体执行完毕，控制即转到 if 语句的下一条语句；如果结果为 False，则跳过条件语句体，控制直接转到 if 语句的下一条语句。图 3.4 中的流程 图形象地解释了 if 语句的语义，其中菱形框表示条件测试。虽然 if 语句根据条件表达式计算 结果的不同而有两个分支，但我们习惯说这种形式的 if 语句实现的是单分支控制结构，因为 有一个分支什么也不做。注意，无论条件是真是假，最后控制都转到 if 语句的下一条语句， 也就是说这条 if 语句内部虽有两个分支，但总体只有一个出口①。

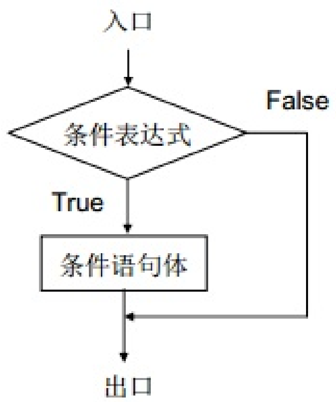


图 3.4 单分支控制结构

利用单分支形式的 if 语句，可以很容易地改进程序 3.1，使之具有高温告警功能。

> ① 在标准流程图符号中有一种连接符号，用于将两个进入的流程线合并成一个出去的流程线，这里的 if 语句 就可以用连接符号来合并两个分支的末端，形成唯一出口。但为了流程图的简明，我们没有用连接符号，而 是直接将两个流程线合并，相信这并不会影响对流程的理解。

【程序 3.2】eg3_2.py

```
f = input("Temperature in degrees Farenheit: ")
c = (f – 32) * 5.0 / 9
print "Temperature in degrees Celsius:", c if c > 35:
print "Warning: Heat Wave!"
```

这个新版本在原来版本的最后增加了一条 if 语句，该语句的语句体是有条件地执行的。就是说，程序的执行结果取决于变量 c 的值。

我们还可以进一步改进程序 3.2，使之针对极端寒冷的情况也发出寒潮告警信息。具体改 法和上面是类似的，只需再增加一条 if 语句来检查温度是否足够低。

【程序 3.3】eg3_3.py

```
f = input("Temperature in degrees Farenheit: ")
c = (f – 32) * 5.0 / 9
print "Temperature in degrees Celsius:", c if c &gt;= 35:
print "Warning: Heat Wave!" if c &lt;= -6:
print "Warning: Cold Wave!"
```

3.2.2 两路分支结构

有时我们希望根据条件表达式的不同计算结果（True 或 False），分别执行两个不同的语 句序列，这时可以使用具有两个分支的条件语句形式，即 if-else 语句：

```
if <条件表达式>:
    <if-语句体>
else:
    <else-语句体>
```

if-else 语句的语义是：首先计算条件表达式的值，如果结果为 True，则执行 if-语句体；

如果结果为 False，则执行 else-语句体。无论哪种情况，语句体执行完毕之后，控制都转到 if-else 语句的下一条语句。参见图 3.5 所示的流程图。

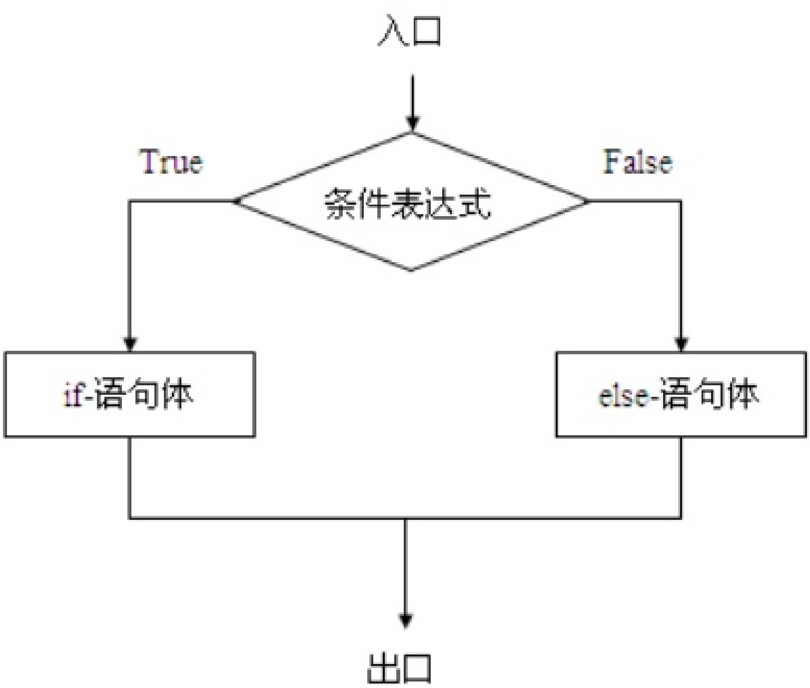


图 3.5 两路分支控制结构

在使用两路分支的 if 语句时要注意：if 部分和 else 部分必须与一对非此即彼的条件相对 应，一个条件为真则另一个条件必为假，反之亦然。例如在程序 3.3 中，c>=35 和 c<=-6 就不是非此即彼的条件，因为还有既非酷热又非酷寒的第三种情形：-6 < c < 35。因此 在程序 3.3 中不能按如下方式使用 if 语句：

```
if c >= 35:
print "Warning: Heat Wave!" else:
print "Warning: Cold Wave!"
```

3.2.3 多路分支结构

如果我们还想进一步改进程序 3.3，使之在-6 < c < 35 的情况下也显示一些信息，这 就需要一个三路的分支结构。三路分支可以利用两个嵌套的 if-else 语句来实现：

```
if c &gt;= 35:
    print "Warning: Heat Wave!"
else:
    if c &lt;= -6:
        print "Warning: Cold Wave!"
    else:
        print "Have fun!"
```

由于 if-else 语句中的<if-语句体>或<else-语句体>可以由任何 Python 语句组成，因此我们 可以再使用一条 if-else 语句，这称为语句的嵌套。分析上面这段代码可知，两条嵌套的 if-else 语句确实实现了三个分支，分别处理 c>=35、c<=-6 和-6<c<35 等三种情形。参见图 3.6， 其中顶层控制结构是一条 if-else 语句，虚线框整体视为它的一个分支；虚线框内是另一条 if-else 语句，它嵌套在顶层条件语句的 else 部分中。

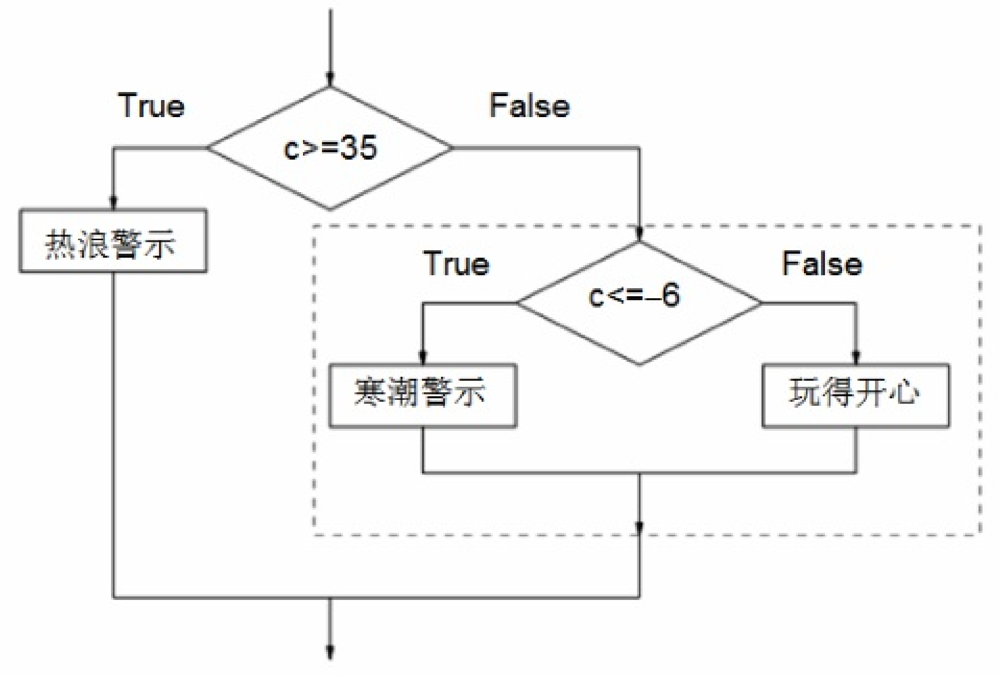


图 3.6 用嵌套 if-else 实现三路分支

用嵌套 if-else 语句虽然能实现三路分支，但并非最好的方法。首先，这种用两个二路分 支来间接实现一个三路分支的做法使得三个分支不在一个层次上，不太符合原问题中的三个 并列分支的题意。其次，这种做法不适合需要更多分支的问题，例如实现五路分支时就必须 采用四层嵌套的 if-else 结构，这会使程序看上去非常难读，尤其是 Python 所要求的下层结构向右缩进的特点会使这条嵌套语句在水平方向占据过宽的空间，导致代码更加难读。

Python 中有一个更好的做法来写多路分支的条件判断，即 if-elif-else 语句。这条语句在 形式上其实是将嵌套 if-else 语句中的 else 与后续的 if 合并成了一个 elif 子句，形如：

```
if <条件 1>:
    <情形 1 语句体>
elif <条件 2>:
    <情形 2 语句体>
    ...
elif <条件 n>:
    <情形 n 语句体>
else:
    <其他情形语句体>
```

if-elif-else 语句的语义是：顺序计算每一个条件表达式，找到第一个为 True 的条件，然后执行其下方缩进的语句体，执行完毕再将控制转到整个 if-elif-else 语句的下一条语句；如 果所有条件表达式的计算结果都是 False，则执行在 else 下方缩进的语句体。可见，这种形式 的条件语句实现了 n+1 个分支。另外，else 子句是可选的，但要注意的是，如果省略 else 子句，则整个语句就可能没有符合条件的分支，从而不执行任何语句体。

不难看出，if-elif-else 语句既能像嵌套 if-else 结构一样实现多路分支，又具有各分支并列 的整齐划一的代码形式，这就解决了嵌套 if-else 语句的两个不足之处。下面我们利用 if-elif-else 结构来进一步改进温度转换程序：

【程序 3.4】eg3_4.py

```
f = input("Temperature in degrees Farenheit: ")
c = (f - 32) * 5.0 / 9
print "Temperature in degrees Celsius:", c
if c >= 35:
    print "Warning: Heat Wave!"
elif c <= -6:
    print "Warning: Cold Wave!"
else:
    print "Have fun!"
```

3.3 异常处理

一个程序即使没有任何语法错误，即使解题的逻辑也正确，在执行的时候仍然可能出现 各种“运行时错误”，导致程序无法按照预定的步骤顺利执行、正常结束。其后果是要么由系 统强行中止程序的运行，要么程序带着错误继续运行而得出错误的结果。这类运行时错误称 为异常或例外（exception）。产生异常的原因是复杂而多样的，既有程序设计的问题，也有运 行环境的问题，如除数为零、用户输入数据的类型或个数不对、列表索引越界等等。

如果一个程序很容易受到异常的影响而崩溃（即中止执行），那就不是好的程序，因为程 序崩溃意味着无法完成预定的计算，不能满足用户的需求。另外，程序崩溃时系统一般会输 出一堆错误消息，这些消息对程序员来说没啥大不了，但对普通用户来说则是难以理解的一 堆技术术语。用户不知道发生了什么，也不知道该如何处理。

因此，程序员必须在程序中加入处理错误的代码，以便在发生错误的情况下能自己处理 错误，使程序错误对用户是不可见的。这样的程序在发生错误的情况下也能正常结束而非崩溃，并且显示给用户的也是可理解的友好的信息。我们称这样的程序是健壮的（robust）。

本节介绍在程序中处理错误的两种方法：一种是传统的错误检测，一种是更现代的异常 处理（exception handling）机制。

3.3.1 传统的错误检测方法

如何提高程序的健壮性？关键显然在于如何发现运行时错误并加以处理。顾名思义，运行时错误是在程序运行时才暴露的，很难在静态的编译阶段检查出来。传统编程方法中常利 用 if 语句来检测可能导致异常发生的条件，以期发现并处理错误。具体的检测方式有两种， 一种是在执行任务之前检测条件，另一种是执行任务之后检测返回状态码或错误码。

作为例子，我们来编写一个求解一元二次方程的程序。利用初等代数知识，我们知道一 元二次方程 ax2+bx+c=0 的两个根是：

$$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

据此很容易写出下面这个程序：

【程序 3.5】eg3_5.py

```
import math
a, b, c = input("Enter the coefficients (a, b, c): ")
discRoot = math.sqrt(b * b - 4 * a * c)
root1 = (-b + discRoot) / (2 * a)
root2 = (-b - discRoot) / (2 * a)
print "The solutions are:", root1, root2
```

本程序先由用户输入一元二次方程的三个系数，然后利用公式算出两个根，并显示结果。 这个版本看上去很直接了当，似乎符合预期的功能，但实际上这个版本很有问题。下面我们 来运行这个程序：

```
>>> import eg3_5
Enter the coefficients (a, b, c): 1,2,3
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module> import eg3_x
File "eg3_x.py", line 3, in <module> discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

由于用户输入的系数 1、2、3 使得一元二次方程的判别式 b2 - 4ac 小于零，因此当程序 运行到调用 math.sqrt 函数时导致错误，程序崩溃并输出上面这一堆错误信息。作为专业的程 序员，对这里发生的一切自然能理解，但作为普通的用户，看到这些天书般的的错误信息时 除了抱怨程序不好用，还能怎么办呢？

为了增强程序 3.5 的健壮性，可以用 if 语句来检查判别式的值，以便区别处理方程有实 数根和无实数根的两种情形，避免在无实数根的情况下崩溃。改进版本如下：

【程序 3.6】eg3_6.py

```
import math
a, b, c = input("Enter the coefficients (a, b, c): ")
discrim = b * b - 4 * a * c
if discrim &gt;= 0:
    discRoot = math.sqrt(discrim)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)
    print "The solutions are:", root1, root2
else:
    print "The equation has no real roots!"
```

从程序中可见，仅当判别式 discrim 大于等于 0 时，才去调用 math.sqrt 函数求其平方根， 这样 sqrt 不会出错，从而避免了程序崩溃；当 discrim 为负数时，并不调用 sqrt，而是向用户显示一些信息，告诉用户发生了什么，程序同样能正常结束。

下面分别测试程序 3.6 对两种情形的判别式的执行效果：

```
>>> import eg3_6
Enter the coefficients (a, b, c): 1,2,3
The equation has no real roots!
>>> reload(eg3_6)①
Enter the coefficients (a, b, c): 1,3,2
The solutions are: -1.0 -2.0
```

从结果可见程序 3.6 确实达到了预期的目的，健壮性得到了增强。

像程序 3.6 这样利用 if 语句来检测可能的出错条件，以阻止可能导致错误的语句的执行， 这是一种常用的错误检测方式。下面介绍另一种错误检测方式。

很多时候要执行的语句实际上是函数调用②，被调用的函数可能是我们自己写的，也可 能是标准函数库里定义的。函数作为一个具有相对独立性的程序块，一般都有自己的错误检 测代码，并根据执行是否正常而返回不同的“错误码”给调用者。这样，函数的调用者可以 无条件地调用函数，然后根据函数返回的错误码来了解函数的执行情况，并基于此来决定下 一步行动。例如，假设有一个求平方根的函数 robustSqrt 在参数为负数时返回错误码-1（由 于实数的平方根总是正数，返回-1 就表明发生了异常）：

```
def robustSqrt(x):
    if x < 0:
        return -1
    else:
        return math.sqrt(x)
```

那我们就可以不必先检测判别式的正负，而是直接调用 robustSqrt，并通过它的返回值来检测是否发生了异常。示例代码片段如下：

```
discRoot = robustSqrt(b * b – 4 * a * c)
if discRoot < 0:
    print "The equation has no real roots!"
else:
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b – discRoot) / (2 * a)
    print "The solutions are:", root1, root2
```

与程序 3.6 中的错误检测代码相比，上面这种错误检测代码更可取。理由是：函数就像 一个提供特定功能的“黑盒”，我们只需调用其功能，不需了解其内部细节，因此让函数自己 在内部进行错误检测更符合“黑盒”原则。程序 3.6 中的错误检测建立在对函数 math.sqrt 内 部执行细节（即负数导致崩溃）的了解之上，因而不符合“黑盒”原则。

> ① reload 函数用于重新运行一个已成功导入的模块。
>
> ② 关于函数，详见第 4 章。

3.3.2 传统错误检测方法的缺点

传统的错误检测方法是过去广泛使用的，这种做法有一个缺点：由于需要检测错误的地方非常多，最终导致程序中充斥着大量的错误检测代码，这些“喧宾夺主”的代码使得程序 控制结构复杂，程序逻辑难以理解，代码也难维护。例如，如果每次调用函数都要检测其返 回的错误码，会导致程序中存在大量如下形式的代码片段：

```
x = doOneThing()
if x == ERROR:
    异常处理代码
    ......
```

或者更简练（但更难读）地写成：

```
if doOneThing() == ERROR:
    异常处理代码
    ......
```

假如我们解决某个问题的算法是顺序执行三个步骤，用三个函数调用表示如下：

```
doStep1()
doStep2()
doStep3()
```

这段代码清晰地表明了要做的事情是什么，逻辑非常容易理解。但是当我们加入大量的

错误检测代码之后，可能写出如下代码：

```
if doStep1() == ERROR：
    错误处理代码 1
elif doStep2() == ERROR:
    错误处理代码 2
elif doStep3() == ERROR:
    错误处理代码 3
```

从这段代码可见，原先很清晰的连续的三个步骤与错误检测代码纠缠在一起，导致解决 问题的关键算法变得非常隐晦。当需要检测的异常情形（对应着函数返回的错误码）很多的 时候，程序逻辑会深深地掩埋在这些错误检测代码之中。

3.3.3 异常处理机制

那么，有没有办法使我们既能增强程序的健壮性，又不影响程序逻辑的清晰和完整呢？

现代编程语言提供了异常处理机制来解决这个问题。异常处理机制的基本思想是：程序运行时如果发生错误，就“抛出”一个异常，而系统能够“捕获”这个异常并执行特定的异常处 理代码。图 3.7 中给出了异常抛出和捕获的示意图，从图中可见，异常实际上是一种可能改 变程序控制流的事件，使我们能跳出某个正常执行的程序块。

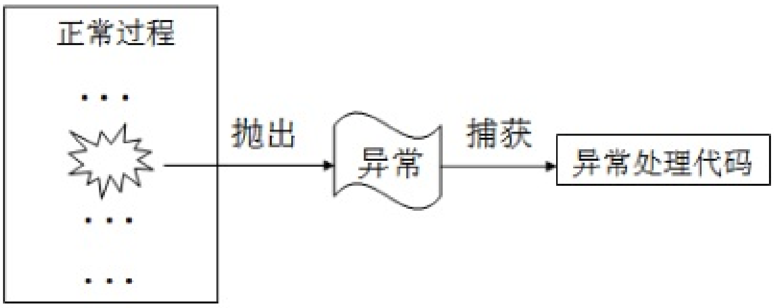


图 3.7 异常的抛出、捕获和处理

打个比方，当厨师在按照预定的菜谱做菜时，如果执行到某个步骤发现酱油没了或炉具 坏了，就只能跳出正常步骤，转到能处理这种意外的程序：酱油没了可以去买酱油，买回来 后可以继续做菜；炉子坏了一般只好中止做菜。

Python 语言也提供了这样的异常处理机制。在 Python 中，异常处理是通过一种特殊的控 制结构来实现的，即 try-except 结构。try 语句的最简单形式如下：

```
try:
    <语句块>
except:
    <异常处理语句块>
```

其语义是：执行<语句块>，如果一切正常，执行结束后控制转向 try-except 的下一条语句；如果执行过程中发生了异常，则控制转向异常处理语句块，执行结束后控制转向 try-except 的下一条语句。

缺省异常处理

我们前面所写的程序都没有使用异常处理，这时如果程序出现运行时错误，实际上会由 Python 进行缺省的异常处理。Python 所做的事情只是简单地中止程序运行，并显示一些错误信息。例如：

```
>>> a = "Hello"
>>> print a[5]
Traceback (most recent call last): File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

上面第二条语句导致索引越界错误，这个异常被 Python 捕获并显示标准错误信息。从例 中可见，错误信息包括两个部分：错误类型（如 IndexError）和错误描述（如 string index out of range），两者用冒号分隔。另外，Python 还追溯错误发生的地方，并显示有关信息。

程序自己处理异常

Python 的缺省异常处理使应用程序中止，控制转给 Python 解释器。如果应用程序需要在 发生异常的情况下仍能正常结束，就需要使用 try-except 语句来自己捕获并处理异常。例如：

```
>>> a = "Hello"
>>> try:
        print a[5]
    except IndexError:
        print "Index wrong!"
Index wrong!
```

索引越界错误发生之后，控制自动转到 except 子句下面的处理代码，处理完毕还可以继 续执行程序的其他语句。如果没有错误，则忽略 except 部分。

异常处理机制的优点

相对于错误检测代码，使用异常处理机制可以使程序的核心算法代码与错误处理代码相互分离，从而保持程序结构的清晰。如果要了解程序的主要算法，只需读 try 下面的语句块， 完全不会被繁杂的错误检测打扰。例如，如果用 try-except 语句来实现上一小节中的“三步 走”例子，只需用一个 except 子句来捕获 doStep1、doStep2 和 doStep3 等步骤可能抛出的各 种异常，代码形如：

```
try:
    doStep1()
    doStep2()
    doStep3()
except:
    doErrorProcessing()
```

显然，这种形式的代码既能保持算法逻辑的清晰完整，又能实现错误检测，圆满解决了 上一小节中提到的错误检测的弊端。

如果要做的事情步骤很多、流程很复杂，将所有代码堆积在 try 之下又会使程序结构不 清晰。这时可以利用模块化设计将程序逻辑表达为许多函数①，然后在 try 部分调用各函数， 形如：

```
def doMyJob():
    doStep1()
    doStep2()
    ...
    doStep100()
    try:
        doMyJob()
    except:
        doErrorProcessing()
```

也可以让程序的每个模块各自具有自己的异常处理，而不是将异常抛出给其他模块处理。

分类处理异常

以上用到的简单形式的 try 语句不加区分地对所有错误进行相同的处理，如果需要对不同错误类型进行不同的处理，则可使用更精细的控制：

> ① 见第 4 章。

```
try:
    <语句块>
except <错误类型 1>:
    <异常处理语句块 1>
    ...
except <错误类型 n>:
    <异常处理语句块 n>
except:
    <缺省异常处理语句块>
```

其语义是：执行<语句块>，如果一切正常，执行结束后控制转向 try-except 的下一条语句；如果执行过程中发生了异常，则系统依次检查各个 except 子句试图找到与所发生的异常相匹配的错误类型。如果找到，就执行相应的异常处理语句块，如果找不到则执行最后一个 except 子句下的缺省异常处理语句块。异常处理结束后控制转到 try-except 的下一条语句。注意， 最后一个不含错误类型的 except 子句是可选的，用于捕获所有未预料到的错误类型。如果未 使用最后这个 except 子句，那么当异常与所有错误类型都不匹配时，则由 Python 解释器捕获 异常并处理之①。如前所述，Python 的缺省异常处理是中止程序并显示错误信息。

理解了异常处理的基本知识后，下面我们利用 try-except 语句来改写一元二次方程求解 程序，代码如下：

【程序 3.7】eg3_7.py

```
import math
try:
    a, b, c = input("Enter the coefficients (a, b, c): ")
    discRoot = math.sqrt( b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)
    print "The solutions are:", root1, root2
except ValueError:
    print "The equation has no real roots!"
```

程序 3.7 这个版本和程序 3.5 中的版本非常相似，只是在程序 3.5 所示的核心算法之外增 加了一个 try-except 结构。从而做到了既保持清晰的核心算法逻辑，又避免因判别式为负数 而导致程序崩溃。让我们再次以系数 1、2、3 来执行这个程序：

```
>>> import eg3_7
Enter the coefficients (a, b, c): 1,2,3
The equation has no real roots!
```

可见不适当的系数并没有使程序崩溃，异常处理代码捕获了 math.sqrt 引起的异常，使程 序得以正常结束。

除了判别式为负导致 math.sqrt 出错之外，还有多种可能导致程序出错的情形。例如：用 户输入系数的个数不足或者输入的是字符串而非数值均可导致 TypeError，输入未定义的变量而非字面值可导致 NameError，为系数 a 输入 0 可导致 ZeroDivisionError，等等。使用 try...except 语句可以捕获任何预先想到的异常类型，使用缺省 except 还可以捕获所有未预料到的异常， 从而使程序在任何运行时错误发生的情况下都不会崩溃。下面是更完善的解方程程序版本：

> ① 对于多层的程序结构（外层调用内层，内层又调用更内层），当发生异常时，如果本层没有匹配的异常处 理代码，则该异常被交给上一层处理。上一层没有匹配的异常处理代码就继续往上传，直至要么找到匹配， 要么到达顶层（即 Python 解释器）进行缺省异常处理。

【程序 3.8】eg3_8.py

```
import math
try:
    a, b, c = input("Enter the coefficients (a, b, c): ")
    discRoot = math.sqrt( b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)
    print "The solutions are:", root1, root2
except ValueError:
    print "The equation has no real roots!"
except TypeError:
    print "Wrong coefficients!"
except NameError:
    print "Undefined variable!"
except:
    print "Something wrong!"
```

总之，使用了 try 语句后，不管发生什么错误（除了 Python 系统之外的问题，如操作系 统错误、硬件故障等）程序都可以避免崩溃。

使用 try-except 语句尽管看上去有点繁琐，但它确实是编写健壮程序所必需的。在实际 应用开发中，要想写出职业水准的程序，就应该考虑各种可能的异常情形，以防止用户得到 难以理解的结果。当然，初学编程时，经常不去考虑错误输入等程序健壮性问题，而是把注 意力放在算法和数据结构等方面。

3.4 循环控制结构

计算机是以一步一步执行指令的方式来解决问题的，程序员要做的事情就是将问题的解决方案表达成一步一步执行的指令序列。在解决问题的指令序列中，经常会遇到需要重复执 行的一组操作。例如，假设程序要求用户输入 5 个数据，怎么表达这个要求呢？一种方式是 将所有步骤罗列出来：

```
Step1：输入 1 个数据存入变量 a
Step2：输入 1 个数据存入变量 b
Step3：输入 1 个数据存入变量 c
Step4：输入 1 个数据存入变量 d
Step5：输入 1 个数据存入变量 e
```

这种表达方式既直接又简单，但是明显有局限性——若要求输入 100 个数据怎么办？难道直

接用 100 行几乎相同的指令，并且命名 100 个变量来存储输入数据？这显然是非常笨拙的编程方式。我们来看另一种表达方式：

```
Step1：输入 1 个数据存入集合 a
Step2：如果已经输入了 5 个数据，就结束；否则转到 Step1。
```

这种表达方 式一方面使 用了集合来 存储大量数 据，另一方 面采用了“ 循环”结 构（Step1®Step2®Step1）来控制流程。其功能与罗列所有步骤的方式是一样的，但形式上更简洁，并且可以轻松地推广到 100 个输入的情形而不增加代码量（只需将 Step2 中的 5 改成 100 即可）。鉴于编程语言的任务之一就是提供合适的语言构造使程序员能够方便地表达程序 逻辑，这第二种表达方式应该被编程语言所支持，事实上也正是如此。例如对于大量数据的 存储，Python 提供了列表等类型，只要一个变量就能存储 100 个数据。而为了表达重复执行的指令，Python 提供了循环语句，这正是本节要介绍的主要内容。 循环是程序中的一组语句，只写一次但可以连续执行多次。在编程语言中，构成循环的

这组语句的连续执行的次数一般有三种方式指定：第一，直接指定循环次数；第二，遍历一个数据集合，从而间接指定循环次数（集合有多少成员就循环多少次）；第三，指定一个条件， 当条件满足时循环或者循环执行到条件满足为止。

下面介绍 Python 语言中的循环结构。

3.4.1 for 循环

最简单的循环是已知重复执行次数的循环。小学生经常有这样的“痛苦”时刻：因为一 个字（比如“烦”）写错了，被老师要求订正 10 遍。这时小学生没有捷径可走，只能在本子 上一遍一遍地写上 10 次。如果是命令计算机在屏幕上写 10 遍“烦”，是不是也只能用下面的 10 行指令来实现呢？

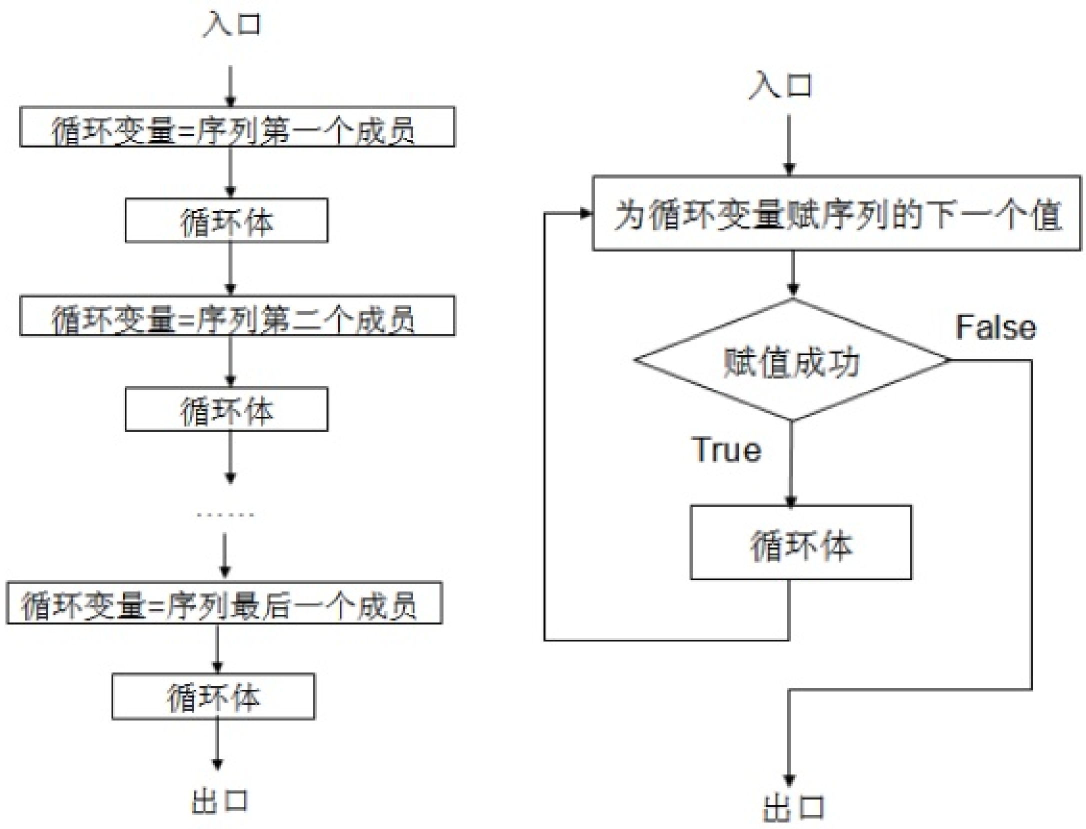
```
print "烦"
print "烦"
......
print "烦"
```

显然，这种做法非常烦琐，需要在键盘上敲很多键，而且不具有扩展性（抄写 1 万遍怎么办？）。本节介绍 Python 语言中的 for 语句，可以很好地解决上面这个问题。

for 语句的常用语法形式如下：

```
for <循环控制变量> in <序列>:
    <循环体>
```

其语义是：用序列中的成员逐个赋值给循环控制变量，对每一次赋值都执行一遍循环体。当序列被遍历，即每一个值都用过了，则循环结束，控制转到下一条语句。注意，循环体部分相对于 for 部分要左缩进。for 语句的执行流程如图 3.8（a）所示，或更常见地画成图 3.8（b） 的样子。



（a）（b）

图 3.8 for 循环的流程图

for 循环的循环次数显然是由序列中有多少成员（即序列长度）决定的，而循环控制变量 的作用是存储每一次循环所涉及的序列成员的信息。不难理解，循环控制变量的作用一般仅 仅限于这个循环语句，出了这个循环语句循环控制变量就失去了它的作用。因此，编程序时 最好用一个新变量来做循环控制变量，而不是用前面已经使用过的变量名，以便突显循环控 制变量专用于循环控制的角色，不引起理解上的混乱。

下面介绍利用 for 语句建立的几种常见循环模式。

计数器循环

我们可以用下面的 for 语句来解决将“烦”字显示 10 遍的问题：

```
>>> for i in range(10):
print "烦",
烦 烦 烦 烦 烦 烦 烦 烦 烦 烦
```

这条 for 语句的执行流程是这样的：计算 range(10)得到列表[0,1,2,3,4,5,6,7,8,9]，然后令 变量 i 从左往右取遍列表[0, 1, ... , 9]中的每一个值，并对所取的每个值都执行一次 print 语句。由于列表有 10 个成员，故 print 就被执行了 10 次。注意 print 语句相对上面一行要缩进，表示它是 for 循环要重复执行的语句。

在上面这个例子中，循环体与循环控制变量的值没有关系，即不管循环控制变量从序列 中取的值是什么，循环体总是固定地执行 print "烦"。从效果上看，循环控制变量和序列仅仅 起着计数器的作用，用于控制循环次数，这种循环模式称为计数器循环。

当然，循环体引用循环控制变量的值也是很常见的。这种情况下，循环控制变量不仅控 制循环次数，而且直接影响循环体的行为。例如，下面这个程序可以计算 1 到 n 的平方和：

【程序 3.9】eg3_9.py

```
n = input("Input a number: ")
sum = 0
for i in range(1,n+1):
    sum = sum + i*i
print "The result is:",sum
```

执行此程序，将得到如下输出：

```
>>> import eg3_9
Input a number: 100
The result is: 338350
```

遍历数据项列表

for 语句是针对任意序列进行遍历来建立循环的，并非只能与 range(10)之类的数字序列 搭配构成计数器循环。例如，下面的代码针对一个杂乱数据项构成的列表进行遍历：

```
>>> data = ['Born on:','July',2,2005]
>>> for d in data:
...     print d,
...
Born on: July 2 2005
```

这里，数据列表的作用显然不是循环计数。当然，这种对数据列表的数据项进行遍历的 循环也可以转化成对数据列表的索引进行循环，代码如下：

````
>>> data = ['Born on:','July',2,2005]
>>> for i in range(len(data)):
        print data[i],
Born on: July 2 2005
``` 显然这是不必要的。一般来说，直接对数据列表进行循环不但代码简单，执行效率也比针对 列表索引建立的循环要高

下面我们看一个最好用列表索引来建立循环的例子：假如我们希望对 data 列表间隔着访 问其成员（比如每次跳过两个
````

> > > data = ['Born on:','July',2,2005] for i in range(0,len(data),3): print data[i], Born on: 2005 ```

遍历列表的同时修改列表

另一个常见的需要用序列索引来建立循环的情形是在遍历一个列表的同时要修改它，例

如：将列表中的每一个值都加 1。下面这个做法是错误的：

```
>>> data = [1,2,3,4,5]
>>> for x in data:
        x = x + 1
>>> data
[1, 2, 3, 4, 5]
>>> x
6
```

原因是循环体中修改的是循环控制变量 x 而非列表 data。尽管 x 的值来自列表，但修改 x 的

值并不会导致该值的来源处发生改变。为了修改遍历的列表，可以使用列表索引来对列表的相应位置赋值。代码如下：

```
>>> data = [1,2,3,4,5]
>>> for i in range(len(data)):
        data[i] = data[i] + 1
>>> data
[2, 3, 4, 5, 6]
>>> i
6
```

遍历其他序列类型
```

回顾第 2 章的内容，序列是由若干数据项组成的一个有序的集合体，列表、字符串和元 组都是序列。前面的例子中所用的序列都是列表，下面通过例子演示利用字符串或元组建立 循环。先看针对字符串的 for 循环，其作用是将一个字符串的每个字符分拆显示：

```
>>> for c in "Hello World!":
        print c,
H e l l o W o r l d !
```

可见，字符串其实就是一个字符序列，for 语句通过取遍字符串中的每一个字符来建立循环。注意，如果 for 的循环体只有一行语句，那么可以直接跟在 for 那一行的冒号后面。还要注意 print 语句末尾的逗号，它使 print 不换行，从而让各字符显示在同一行上。

再看一个针对元组的 for 循环例子：

```
>>> for i in (1,2,3):
        print i
1
2
3
```

可见，用于 for 循环时，元组和列表具有完全一样的作用。 我们还可以构造更复杂的嵌套结构的序列用于 for 循环，如元组的元组、元组的列表、

字符串的列表等等。以“元组的列表”为例，即列表中每个成员是元组。由于控制循环的循 环控制变量每次取序列中的一个成员作为值，所以这种情况下循环控制变量所取的值是元组。例如：

```
>>> for t in [(1,2),(3,4),(5,6)]:
        print t,t[0],t[1]
```

也可以用多个循环控制变量构成元组来建立 for 循环：

```
>>> for (x,y) in [(1,2),(3,4),(5,6)]:
        print x,y
1 2
3 4
5 6
```

此例中，第一次循环时执行的赋值是(x,y) = (1,2)，亦即 x 和 y 分别赋值 1 和 2。 最后看一个更复杂的序列：

```
>>> for ((a,b),c) in [([1,2],3),['XY',6]]:
        print a,b,c
1 2 3
X Y 6
```

这个 for 语句的第一次循环相当于先执行了赋值：

```
((a,b),c) = ([1,2],3)
```

第二次循环相当于执行了赋值：

```
((a,b),c) = ['XY',6]
```

从此例可见，元组、列表、字符串三种序列类型是非常相似的，可以相互赋值。

3.4.2 while 循环

for 循环要求预先确定循环的次数，但有很多问题难以预先确定循环次数，只知道在什么 条件下需要循环，这时可以使用 while 语句。Python 语言中 while 语句的常用格式是：

```
while <布尔表达式>:
    <循环体>
```

其语义是：当布尔表达式计算为 True 时，执行一遍循环体，执行完毕控制转回 while 语句的开始处重新测试布尔表达式；当布尔表达式计算为 False 时，控制转向下一条语句。注意，循环体部分相对于 while 部分要左缩进。while 语句的流程图如图 3.9 所示。

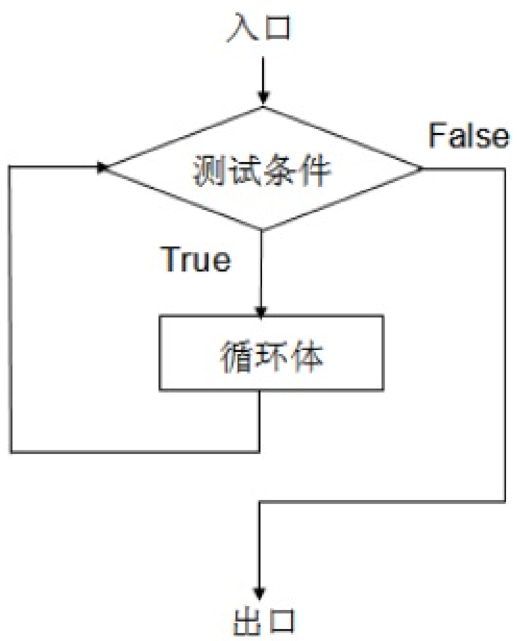


图 3.9 while 循环的流程图

显然，while 语句的循环次数取决于布尔表达式何时变成 False，而不是预先确定循环次 数。稍微深入思考一下就会发现一个问题：万一布尔表达式永远不会变成 False 怎么办？如 果进入循环语句时布尔表达式计算为 True，而循环体又不会影响布尔表达式的值，那么执行 完循环体后控制回到循环入口处时，布尔表达式仍然为 True。这种情况下 while 循环永远不 会停下来，称为无穷循环。程序中如果有一个无穷循环就意味着程序无法终止，我们常说程 序进入了“死循环”，这是程序设计中经常出现的错误。要想避免无穷循环，必须使循环体对 布尔表达式的值产生影响，例如改变布尔表达式中所用到的变量的值。

在实际编程中，while 循环有一些常用的套路或称模式，值得读者熟记于心。下面通过“对 一批数据求和”的例子来介绍这些循环模式。

交互式循环

考虑这样的应用：用户不断输入数据，程序得到数据后不断累加，最后算出输入数据的 总和。显然，这是一个“输入——累加”的反复循环的过程。由于用户输入数据的个数不是 预先给定的，故无法用 for 循环来实现，而 while 语句则能轻松解决这个问题。为了对循环进 行控制，我们每次循环前都询问用户是否还有新数据。这种通过与用户进行交互来决定是否 需要循环的模式称为交互式循环，可以用伪代码表达如下：

```
将循环控制变量 moredata 初始化为"yes"
while moredata 值为"yes":
    获得用户输入的下一个数据
    处理该数据
    询问用户是否还有输入数据，为变量 moredata 赋值
```

下面是利用交互式循环实现的完整程序。

【程序 3.10】eg3_10.py

```
sum = 0 moredata = "yes"
while moredata[0] == "y":
    x = input("Input a number: ")
    sum = sum + x
    moredata = raw_input("More numbers? (yes/no) ")
print "The sum is", sum
```

下面是这个程序的执行情况：

```
Input a number: 2
More numbers? (yes/no) yes
Input a number: 5
More numbers? (yes/no) yeah
Input a number: 8
More numbers? (yes/no) no
The sum is 15
```

要说明的是，这个程序通过检查 moredata[0]来控制是否循环，因此只要用户输入的首字 母是"y"就能进入循环体执行，而任何非 y 开头的输入都导致停止循环。

此版本有个不好的地方：用户需要不停地先回答是否还有数据，有的话再输入数据。这 对用户来说有点烦琐。也就是说，交互式循环其实并不适合“输入 n 个数据求和”的问题。

哨兵循环

解决“输入数据求和”问题的更好方法是使用哨兵循环，即不断执行“输入——累加” 这个循环体，直至遇到一个称为“哨兵”的特殊数据值。任何值都可以当作哨兵，关键是它 必须与正常数据值相互区别。哨兵循环的一般模式如下：

```
前导输入
while 不是哨兵：
    处理数据
    循环尾输入
```

首先，在循环开始之前需要利用“前导输入”获取第一个数据。如果该数据是哨兵，则 不会进入循环，控制转向 while 的下一条语句；如果是正常数据，则进入循环处理之。在循 环体的末尾读取下一个数据，并转到循环开始处的哨兵测试。如此周而复始，直至遇见哨兵 循环才终止。

注意，哨兵循环用到了两条一模一样的输入语句：一条是位于循环体之前的“前导输入”， 另一条是位于循环体末尾的“循环尾输入”。这两条输入语句缺一不可：少了前导输入则无法 进入循环；少了循环尾输入则无法读取下一数据，从而循环一直在对第一个数据进行处理， 导致无穷循环。

使用哨兵循环，需要选择一个特殊数据作为哨兵。这个特殊数据一般和正常数据属于同 样的类型，以便能被 while 语句统一检测。如果哨兵数据和正常数据属于不同类型，那么 while 语句的条件表达式就会变复杂，因为需要处理两种类型的数据。具体选择什么数据作为哨兵，要看具体的应用场景。例如，假设我们输入的数据是考试成绩（非负数），那么可以选－1 作为哨兵，因为负数是不可能成为合法考试成绩的。又假如我们输入的数据是人名（字符串）， 那么可以选空串作为哨兵。一个很常用的场景是文件处理，即输入的数据来自一个文件，这 时可以很自然地在文件末尾存放一个特殊数据作为哨兵，很多语言甚至提供专门的测试文件 尾的手段（如常量 EOF①或函数 eof()之类）。关于文件处理详见第 6 章。

下面我们用哨兵循环来实现求和程序。

【程序 3.11】eg3_11.py

```
sum = 0
x = input("Input a number (-1 to quit): ")
while x &gt;= 0:
    sum = sum + x
    x = input("Input a number (-1 to quit): ")
print "The sum is", sum
```

可见哨兵循环与交互式循环不同，不需要用户不停地回答 yes 来处理数据。下面是此程序的执行情况：

```
Input a number (-1 to quit): 2
Input a number (-1 to quit): 5
Input a number (-1 to quit): 8
Input a number (-1 to quit): -1
The sum is 15
```

如果程序 3.11 中待求和的数据是任意实数的话，那么－1 可能是正常数据，不能作为哨

兵。事实上，所有实数都不能作为哨兵。这时可以采用字符串类型来解决问题，因为正常的输入数据（正数、负数和 0）都可以表示为由阿拉伯数字组成的非空字符串，从而包含非数字字符的字符串都可以用作哨兵。最简单最常用的特殊字符串是空字符串""（注意两个引号之间没有东西），当用户在输入时直接键入回车，Python 即返回空串。当然，这种做法中对数据的处理要麻烦一点，需要将字符串转换为数值类型以便进行求和计算。下面是以字符串方式输入数值数据的求和程序版本：

【程序 3.12】eg3_12.py

```
sum = 0
x = raw_input("Input a number (&lt;Enter&gt; to quit): ")
while x != "":
    sum = sum + eval(x)
    x = raw_input("Input a number (&lt;Enter&gt; to quit): ")
print "The sum is", sum
```

执行示例如下：

```
Input a number (<Enter> to quit): 2
Input a number (<Enter> to quit): 5
Input a number (<Enter> to quit): -8
Input a number (<Enter> to quit):
The sum is -1
```

此运行示例的第四行中，输入时直接按回车键，导致 Python 将空串赋值给了 x，从而使循环终止。

> ① 意为 End-Of-File

在哨兵循环模式中有一个容易犯错误的地方：当用户的前导输入本身就是哨兵，从而导致循环一次也不执行时，while 后面的语句可能没有预料这种情况，导致无法正确执行。例如，我们将程序 3.11 改成计算平均值，算法基本不变：反复输入数据，在循环中累加数据总和 sum 及数据计数 count，当用户输入哨兵－1 时退出循环并计算平均值。代码如下：

```
sum = 0
count = 0
x = input("Input a number (-1 to quit): ")
while x &gt;= 0:
    sum = sum + x count = count + 1
    x = input("Input a number (-1 to quit): ")
print "The average is", sum / count
```

运行此程序，并且首先输入－1，看看会发生什么？是的，由于未进入循环，sum 和 count 都保持为初始值 0，while 的下一条语句在计算平均值的时候发生了除数为 0 的错误！

后测试循环

前面介绍的 while 循环例子都是“前测试”循环，即先检测循环条件，再进入循环。显 然，如果首次测试条件得到 False，则循环体就会一次也不执行。有些实际应用问题要求循环 体必须至少执行一次，例如“输入合法性检查”问题。用户输入数据，程序检查用户的输入 是否合法：如果合法则程序继续向后执行，否则就回到前面要求用户重新输入，直至输入合 法为止。这种输入合法性检查在程序设计中是非常普遍的，好的程序员应该尽量对用户的输 入进行合法性检查。为了实现输入合法性检查，显然需要先获得用户的输入，然后进入循环 语句，即循环至少会执行一次，最后再测试条件决定是否继续循环。因此，我们需要一种“后 测试”循环结构。

有一些语言提供了 repeat-until 或者 do-while 循环结构来实现后测试循环，其语义分别是 “重复做某事，直至满足某条件”和“做某事，当满足条件时重复”。这种循环结构的流程图 如图 3.10 所示①。

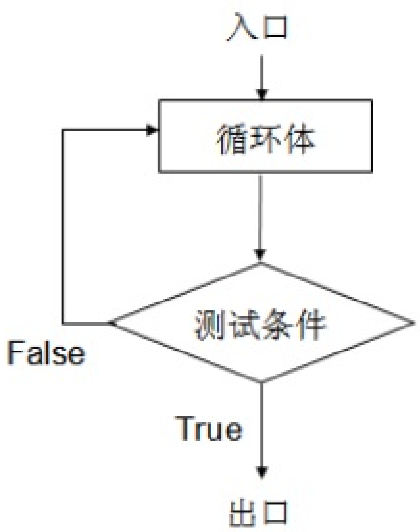


图 3.10 后测试循环控制结构

从图中可见循环体至少执行一次，而循环条件检测位于循环体的最后，这正是“后测试”名称的由来。

> ① 细微差别：这个流程图其实是 repeat-until 结构。将 True 和 False 互换位置，才是 do-while 结构。

Python 语言没有提供类似 repeat-until 结构的语句，但我们不难用 while 来实现后测试循环：只要保证循环条件初始为 True，自然就会执行一次循环体，而后续循环由条件测试决定。 例如，假设程序要求用户输入一个正数，则可用下面的代码片段来检查输入合法性：

```
x = -1
while x < 0:
x = input("Please input a positive number: ")
```

其中第一行的作用是使首次条件检测为真，因此循环体至少执行一次；接下去的条件检测就相当于位于循环体最后，整个语句也就等价于后测试循环。

不难看出，程序 3.10 中的交互式循环将 moredata 初始化为 True，因此实际上是后测试 循环的一种，效果是使用户至少输入一个数据。

while 计数器循环

虽然一般来说 for 语句用于固定次数的循环，while 语句用于不定次数的循环，但两者之 间并无本质不同，完全可以用 while 来实现固定次数的循环。为了循环 n 次，用 while 实现的 计数器循环模式如下：

```
计数器 count 置为 0
while count < n:
    处理代码
    count = count + 1
```

可见，用 while 语句实现计数器循环时需要手动控制循环控制变量 count 的变化，而 for 语句是自动处理的。使用时具体要注意两点：第一，必须为 count 赋初值，否则 while 后的布 尔表达式无法计算；第二，必须在循环体中改变 count 的值，否则会导致无穷循环。

例如，前面罚写 10 遍"烦"字的计数器循环，可以用 while 语句实现如下：

```
>>> i = 0
>>> while i < 10:
        print "烦",
        i = i + 1
烦 烦 烦 烦 烦 烦 烦 烦 烦 烦
```

在此例中，i 初值为 0，以后每次循环都加 1，由于从 0 到 9 都是满足循环条件的，所以 总共执行 10 次循环。第 10 次循环后，i 值为 10，不满足循环条件，故退出循环，控制转到 下一条语句。

上面的 while 计数器循环模式是让计数器从小到大变化，同样常用的还有让计数器从大 到小变化的模式：

```
计数器 count 置为 n
while count > 0
    处理代码
    count = count - 1
```

使用这种模式实现上面的例子，代码如下：

```
>>> i = 10
>>> while i > 0:
        print "烦", i = i - 1
烦 烦 烦 烦 烦 烦 烦 烦 烦 烦
```

值得一提的是，计算机编程中我们经常是从 0 开始计数的，而不是日常生活中的从 1 开始。上例中的 while 语句都是对从 0 到 9 的 10 个值进行循环。如果读者更习惯从 1 开始计数，也可以先为 count 赋予初值 1，然后在循环体中不断加 1，从而实现对从 1 到 10 的 10 个值进 行循环，这样能与日常说的第 1 次、第 2 次、…、第 10 次在序数上对应起来。但是要注意的 是，这时需要将循环条件改成 count<=10 或 count<11。循环控制变量的边界值是初学者容易 犯错的地方，经常导致循环次数多 1 次或少 1 次。

3.4.3 循环的非正常中断

正常的循环总是按“从头到尾再回到头”的方式进行的，但是很多编程语言都提供了在特定条件下打破正常循环方式的语句，目的是在某些情况下可以编写更简单的代码。Python 语言中也提供了这样的语句：break 和 continue。

break 语句

for 或 while 语句的循环体中可以使用 break 语句，其效果是终止本次循环，并将控制跳 出循环语句，转到循环语句的下一条语句。

break 语句经常与一个无穷循环搭配使用，因为按正常途径是跳不出无穷循环的，而用 break 则能以非正常方式跳出循环。例如，我们换一种方法来实现“输入合法性检查”，代码 如下：

```
>>> while True:
        x = input("Please input a positive number: ")
        if x > 0:
            break
Please input a positive number: -2
Please input a positive number: 0
Please input a positive number: 2
>>>
```

这里循环条件是常量 True，它的值是不可能被循环体改变的，即永远为真，所以这是一个无穷循环①。与前面用后测试循环实现的输入合法性检查代码相比较，可以看到这段代码不需要人为设置循环的初始条件为 True，因为循环体总是要执行的。这样的代码更加简单直观，但问题是如何退出无穷循环呢？从上面的代码可见，当用户输入数据不正确，就会不断循环， 要求用户重新输入；当用户确实输入了正数 x，就会执行 break 语句，其作用是跳出循环，控 制转到下一条语句（通常是接着对合法的输入数据 x 进行处理的代码）。

再看一个用 break 跳出 for 循环的例子：

```
>>> for i in range(10):
        print "烦"
        if i > 4:
            break
烦 烦 烦 烦 烦 烦
```

从代码可见，虽然 for 语句本身说的是要罚写 10 遍“烦”字，但循环体中却另有安排：如果 已经抄写了 6 遍（思考：为什么是 6 遍？颠倒循环体中两条语句的顺序又会如何？）“烦”字后，就不耐烦地跳出了循环。

> ① 由于 1 在很多语言中都解释为 True，所以有很多人喜欢用 while 1 来表示无穷循环。

利用无穷循环和 break 搭配的结构同样可以实现前面介绍的哨兵循环，一般模式如下：

```
while True:
    输入下一个数据 x
    if x 是哨兵:
        break
    处理 x
```

与哨兵循环模式相比较，就能看出这种模式不需要循环之前的前导输入，但在循环体中必须用 break 才能退出循环。

continue 语句

for 或 while 语句的循环体中还可以使用 continue 语句，其作用是终止本次循环，并将控 制转到循环语句的开始处，“继续”执行下一次循环。

对 break 与 continue 语句进行比较，可知两者都终止执行当前循环，但接着 break 会跳出 循环语句，而 continue 则继续下一次循环。

看一个简单例子：对数据列表中的奇数求和。算法很简单，只需逐个检查列表中的数据， 如果是奇数就加到总和上，如果是偶数就忽略之，直接去检查下一个数据。代码如下：

```
>>> a = [23,28,39,44,50,67,99]
>>> sum = 0
>>> for i in a:
        if a % 2 == 0:
            continue
        sum = sum + i
>>> print sum
228
```

要说明的是，break 和 continue 语句导致循环结构有多个出口，这不符合结构化编程的基 本思想。虽然使用它们没什么大问题，但仍然建议读者尽量避免使用，尤其是不宜在一个循 环体中使用多个 break 或 continue 语句。关于结构化编程，详见 3.5 节。

3.4.4 嵌套循环

为了实现复杂的算法，控制结构可以相互嵌套，即一个控制结构处于另一个控制结构的 内部。前面我们见过 if 结构的嵌套，现在我们讨论循环的嵌套。

先考虑“一维”数据结构——由简单数据值构成的列表，为了遍历列表以处理其中数据， 我们需要一个循环。例如用一个循环来计算列表中所有数据之和：

```
>>> a = [1,2,3,4,5]
>>> sum = 0
>>> for i in a:
        sum = sum + i
>>> print sum
15
```

但是一个循环不足以解决“二维”数据结构——如矩阵。第 2 章中介绍过，编程语言中 用“列表的列表”来表示矩阵。用一个循环可以每次取列表中的一个值来处理，但这个值本 身又是一个列表，因此又需要一个循环来遍历之。这样我们就得到一个嵌套的循环结构来处 理二维数据结构，如下面的代码所演示的那样：

```
>>> a = [[11,12,13,14],[21,22,23,24],[31,32,33,34]]
>>> sum = 0
>>> for i in a:
        for j in i:
            sum = sum + j
>>> print sum
270
```

可见，为了遍历矩阵，需要由外循环和内循环嵌套来完成：外循环负责对所有的行进行 遍历，而内循环负责对当前行的每一列进行遍历。首先由外循环取一行，再由内循环处理这 一行；当内循环处理完一行，控制又转到外循环去取下一行。例如，外循环控制变量 i 取第 一行[11,12,13,14]时，内循环控制变量 j 取遍 i 中的 11、12、13 和 14 进行处理，处理完毕后 i 再取第二行进行处理，依次类推。

当然，二维数据结构不一定都像矩阵这么整齐，每一行数据可能有长有短，因此在用嵌 套循环来遍历所有数据时，内循环的循环次数常常要根据外循环的循环控制变量值做相应调 整。作为例子，请看下面这个打印乘法口诀表的嵌套循环：

```
>>> for i in range(1,10):
        for j in range(1,i+1):
            print "%dx%d=%-2d" % (j,i,j*i),
        print
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20 5x5=25
1x6=6 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

这段代码虽然很简单，却展示了嵌套循环编程中常用的两个技巧。首先，内循环的循环 次数（由 range(1,i+1)决定）依赖于外循环的循环控制变量 i，因为对 i=1 只有一个乘式，对 i=2 有两个乘式，…，对 i=9 有九个乘式。其次，为了将每个 i 值所产生的乘式放在同一行上，且不同 i 值的乘式放在不同行上，我们在外循环的循环体中与内循环 for 语句并列写了一条 print 语句，以便每当内循环结束就换一次行；而在内循环的循环体中，print 语句的最后是用逗号结尾的，表示每次循环期间不换行。

设计嵌套循环时，一般先设计外循环，这时并不考虑内循环要做的事。当把外循环的结 构搭建好之后，再去设计内循环的任务，这时又不需要考虑外循环。最后将内外两个循环的 代码融合在一起，就得到了完整的嵌套循环代码。

和两重嵌套循环类似，嵌套循环还可以由三重循环构成，用于处理三维数据结构。依此 类推，n 重嵌套循环可用于处理 n 维的数据结构。

3.4.3 节中介绍的 break 语句只能跳出包围它的那一层循环。在嵌套循环结构的情况下， 一条 break 语句虽然跳出了本层循环，但跳不出外层循环，因此控制仍然可能处于某个循环 体中。例如，我们改写打印乘法口诀表的程序，使得一部分乘式不显示。代码如下：

```
>>> for i in range(1,10):
        for j in range(1,i+1):
            if j &gt; 4: break
            print "%dx%d=%-2d" % (j,i,j*i),
        print
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20
1x6=6 2x6=12 3x6=18 4x6=24
1x7=7 2x7=14 3x7=21 4x7=28
1x8=8 2x8=16 3x8=24 4x8=32
1x9=9 2x9=18 3x9=27 4x9=36
```

从上面的代码和结果可以看出，对于内循环所处理的每一行，j>4 的乘式都被 break 跳过了，但是外循环仍能继续执行。

3.5 结构化程序设计

早期的计算机运算速度慢、存储空间小，主要应用于科学计算。因此那时的程序在结构

方面很简单，程序员主要追求的是精细的编程技巧，以期在有限的存储空间内尽快地计算出结果。例如，在用汇编语言编程序时，如果要计算某个数 A 乘以 2，聪明的程序员不会用乘法指令来做这件事，而是会采用左移指令：将 A 的二进制表示左移 1 位（右边补 0）①。这是 因为执行一条乘法指令所需的时间通常是执行一条左移指令所需时间的若干倍。可见，这个 时期的程序设计类似于手工作坊，全凭程序员个人的聪明才智写出高质量的程序。

随着计算机硬件技术的发展，计算机的应用领域越来越广，待解决的问题越来越复杂， 导致计算机软件越来越大型化、复杂化。这时，程序的运行时间和占用的存储空间不再是程 序设计的关注焦点，而软件的开发效率和可靠性取而代之成为程序设计的巨大挑战。高级编 程语言的发明大大提高了编程效率，改善了程序质量，但仍没有解决大型软件开发周期长和 可靠性差的问题，这导致了上世纪 60 年代的所谓“软件危机”。

为了应对危机，计算机科学家对程序设计方法和工具、软件开发全过程的管理和控制等 等课题进行了研究。在程序设计方法方面的研究导致了结构化、模块化、面向对象等方法的 产生，在软件开发过程管理和控制方面的研究则导致一个新学科——软件工程的创立。

在介绍结构化编程思想之前，我们先简单介绍一下按照软件工程的思想该如何开发一个 程序。

3.5.1 程序开发过程

软件工程将软件系统的开发过程划分为前后相继的若干个阶段，称为系统开发生命周期（SDLC），开发人员必须严格遵循 SDLC 来开发软件系统。SDLC 包括分析当前系统、定义新系统的需求、设计新系统、开发新系统、实现新系统和评估新系统等阶段。本书主要关注程序设计，所以下面我们只讨论“开发新系统”这个阶段。

开发新系统阶段的任务大体上就是程序设计，它本身又可划分为几个步骤，构成程序开 发周期（PDC）。PDC 的各个步骤如下：

> ① 如果不理解，可以用四位十进制数 0123 乘以 10 做类比：将 0123 左移一位，右边补零，即得 1230。

- 明确需求：明确问题是什么，理解用户在功能方面的要求。
- 制定程序规格：描述程序要“做什么”。
- 设计程序逻辑：设计程序的解题过程，即描述“怎么做”。
- 实现：使用一种编程语言来实现设计，即编写程序代码。
- 测试与排错：用样本数据执行程序，测试结果是否与预期吻合。如果发现有错误（行 话称为 bug）则排除错误（debug）。
- 维护程序：根据用户需求持续开发、改进程序。 程序规格描述程序的要做什么事情，对于简单程序通常只需要描述程序的输入和输出分别是什么。

设计程序逻辑是核心步骤，其主要任务是设计出满足程序规格的算法，这也是本书自始 至终讨论的重点。在设计阶段，我们经常要使用两种设计工具：程序流程图和伪代码。我们 在前面介绍控制结构时已经通过例子展示了这两种工具的用法。

对于复杂程序，还需要使用其他的工具，如层次图或结构图（参见第 4 章）。 程序逻辑设计好之后，即可用一种编程语言来实现，如本书采用的 Python 语言。常用的

编程语言都是命令式语言，它们用一条一条的命令（语句）组成序列来表达程序逻辑。如何将语句编排在一起，形成结构良好的程序，这正是结构化程序设计要解决的问题。

程序编好之后需要进行测试，以便发现错误并修改程序。测试的方法是，用样本数据去 执行程序，并检查计算结果是否符合预期。对于复杂结构的程序，应当先进行单元测试，最 后进行联合调试。

程序即使已经交付用户投入运行，仍然还有维护问题，以便排除测试调试阶段未发现的 错误，或者根据用户需要升级改进程序。

本书讨论的重点是设计程序逻辑，这个任务完成的好坏，不但直接影响下一阶段的编码 实现，还会影响以后的测试、调试和维护。例如，如果程序结构设计的很乱，程序就难以理解，将来不管是自己还是换人来对程序进行升级改进，都会非常困难。

另外软件开发中还有一件重要的事情，那就是文档化。文档化工作不仅指 PDC 各个阶段 的成果要体现在各种文档中（如设计文档、用户手册、联机帮助等），还包括程序代码中的各种文档化手段（如程序注释）。

3.5.2 结构化程序设计的基本内容

简单问题的求解过程通常是直接了当的，可选择的执行路径不多；但对于复杂问题，一般能设计出多种求解过程。在各种求解过程中，有些过程会比其他过程“好”，当然这个“好” 的意义是依赖于具体问题的。打个比方，为了烧一壶开水，恐怕所有人都会按照“向壶中加 入冷水；壶放到炉子上；点火烧至沸腾”这样的过程来解决。但如果是烧冬瓜排骨汤，则外 行会将冬瓜和排骨一起入锅加水煮；有点经验的人则知道先煮排骨，排骨快熟了才加冬瓜一 起煮；而老练的厨师则会先将排骨焯水，然后再加水煮，排骨熟了才加冬瓜。如果再考虑各 种佐料的使用，显然冬瓜排骨汤的制作过程是多种多样的。哪种制作过程好呢？美食家会告 诉我们厨师的做法是好的，因为按他们的做法能保证排骨熟透而冬瓜不烂，而且焯过水的排 骨更干净并可减少油腻。

至此，一个问题摆在了我们面前：如何设计出能解决特定问题的“好”的程序？为了回 答这个问题，需要先定义什么是“好”程序。一般来说，好的程序不但要能正确地解决问题， 而且还应该是执行效率高、易理解、易维护、可扩展的。

程序设计过去曾被看作是个人技艺，程序的好坏完全依赖于程序员的个人才能。但后来 计算机科学家们认识到，程序设计是一门可以给予科学解释的学问，可以建立良好的设计方 法来指导程序员进行程序设计。普通程序员只要遵循这些设计方法，都能编写出良好的程序。

计算机科学家提出了许多程序设计方法，最早提出也是最基本的一种方法就是这里要介 绍的是结构化程序设计（structured programming，简称 SP）。SP 是以 Dijkstra 为代表的计算机科学家于上世纪 60 年代后期建立起来的，是从程序文本结构的角度来阐述怎样的程序是良好的。SP 的基本思想是要确保程序具有良好的结构，使程序易理解、易验证和易维护。当然， SP 并没有一个严格的、公认的定义，其具体内容大致包括以下几个原则。

只用三种基本控制结构 解决复杂问题时，程序可能需要建立复杂的控制流程，这是不是意味着编程语言应该提供更多的复杂控制结构呢？答案是否定的。计算机科学家证明了所谓“结构化定理”：任何程 序逻辑都可以只用顺序、条件分支、循环这三种基本控制结构来实现。因此，我们在开发程 序时，应该只使用这些基本控制结构，并将它们串联、嵌套在一起，从而搭建出整个程序。 本章前面介绍了条件分支和循环控制结构的多种常见使用模式，读者应当熟练地掌握这些模 式。当遇到复杂问题时，可以利用流程图工具，将复杂的控制流程转化成这些基本控制结构 的串联和嵌套。

goto 语句是有害的

较老的编程语言（如 BASIC、Pascal 和 C 等）中提供了 goto 语句，这条语句的作用是将 控制直接转到程序中的指定位置。使用 goto 可能写出这样的代码：

```
……
ENTRY: count := 0; while count &lt; n do
begin
……
if sthWrong then goto EXIT else goto ENTRY;
end;
EXIT: writeln("End");
……
```

goto 语句看上去用起来很直接、很方便，很多人在设计程序流程遇到麻烦时第一感就会 想用 goto 语句。但是可以想象，如果程序中大量使用 goto 来控制程序的流程，这样的程序 就像一团乱麻，程序的静态结构与动态执行不一致，是非常难理解、难维护的。Dijkstra 首先 提出 goto 语句是有害的，并提出应当编写结构清晰的程序，以使程序易写、易读、易验证和 易维护。

事实上，goto 语句并非必须的语言构造。计算机科学家证明了，使用 goto 的程序一定可 以转化为只包含顺序、条件分支和循环结构的程序，也就是说编程语言中完全可以将 goto 语 句去除。

与 goto 类似的语句还有循环中使用的 break 和 continue 语句，它们都是以跳转的方式将 控制转移到程序其他位置，导致循环有多个出口。按照 SP 的思想，这些语句都应慎用。

单入口单出口的程序块

编程语言的单条语句可以看成是只有一个入口和一个出口，因此前后相继的语句序列构成了单入口单出口的顺序控制结构。而条件控制结构（if 语句）和循环控制结构（for 和 while 语句）的内部虽然可以出现由多条语句构成的语句块，但从外部看同样是只有一个入口和一 个出口（参见图 3.1、图 3.4 等流程图）。总之，基本控制结构（顺序、条件、循环）都是单入口单出口的结构，这种结构具有“可组合”的特性。 如果将两个基本控制结构串联在一起，前一个结构的出口连接后一个结构的入口，那么

所得到的语句序列仍然只有一个入口和一个出口，在效果上完全可以视之为单条语句（见图 3.11）。这就像电子电路中将两个电阻串联后可以视为一个更大的电阻一样。不断重复这个串联过程，将得到由多个控制结构串联而成的结构，它仍然只有一个入口和一个出口，我们称之为程序块。由于程序块只有一个入口和一个出口，在不考虑其内部控制结构的情况下，完全可以将整个程序块视为单条语句，从而可以在不改变其内部控制流的情况下用于程序中任何可以出现语句的地方。

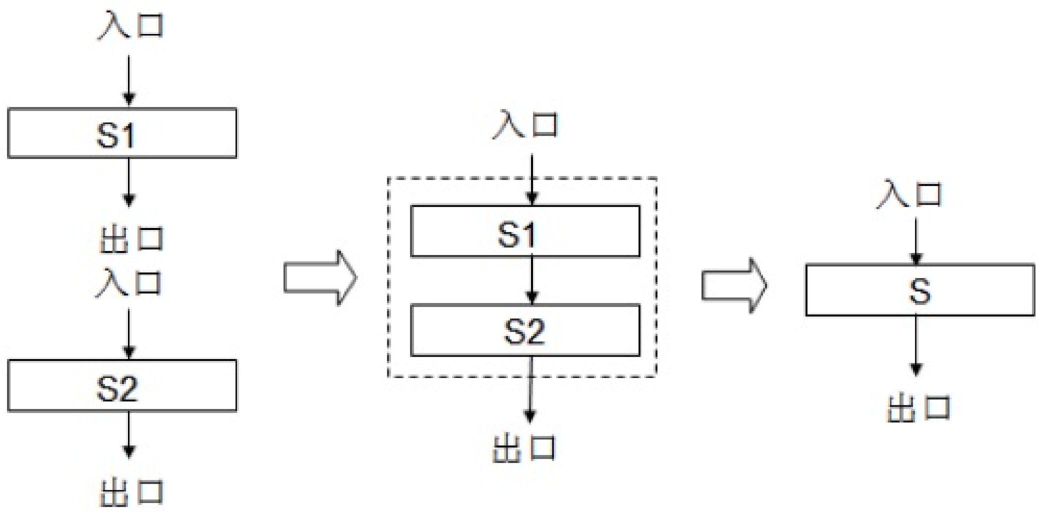


图 3.11 控制结构的串联

除了串联，嵌套也是一种将多个语句组合成一个更大的程序块的形式。例如条件语句的

分支语句体和循环语句的循环体本身都是程序块。结构化程序设计的原则就是利用"单入口单出口"的程序块进行串联、嵌套，最终搭建出复杂程序，这使得程序的结构清晰、层次分明、易理解、易维护。

除了上述几条设计原则，其他如模块化设计、自顶向下逐步求精设计也都是结构化程序 设计的基本内容，下一章对此有详细介绍。

3.6 编程案例：如何求 n 个数据的最大值？

面对复杂问题时，我们需要合理利用基本控制结构，设计出好的算法。对此，并不存在什么机械的套路可循，只能通过大量实践来提供我们的程序设计水平。本节通过一个案例问 题的解决，来展示程序设计过程的挑战性以及“好”程序的特征。

我们要解决的问题是：从 n 个数值中求出最大值。这个问题在实际中很常见——也许不 是作为独立的问题，而是作为其他复杂问题的子问题，因此解决它是很有意义的。我们先来 考虑此问题的一个特例：找出三个数据 x1、x2 和 x3 中的最大值，并把该最大值赋予 max。

3.6.1 几种解题策略

如前所述，对于复杂问题，能够设计出多种多样的算法，并且这些算法各有好坏的不同。

下面我们将对上述最大值问题给出四种解决方法，并讨论每一种策略的好坏。

策略 1：将每个数值与其他两个数值进行比较 由于最大值比其他所有数值都大，所以求最大值的最直接的思路就逐一检查 x1、x2 和x3，看看哪个数值比另外两个数值大。又由于 x1、x2 和 x3 都有可能是最大值，我们可以用 一个三路分支的 if-elif 语句来求解：

```
if x1 >= x2 and x1 >= x3:
    max = x1
elif x2 >= x1 and x2 >= x3:
    max = x2
else:
    max = x3
```

分析一下这条 if 语句，可以看出它用到了两个布尔表达式，而每个布尔表达式又是用 and 联结起来的两个比较运算式，因此可能要经过四次比较运算才能得出最大值。看上去没什么 复杂，但这个算法其实是很不好的。考虑从 4 个数值中求最大值的问题，用这个算法就会需

要 3 个布尔表达式，每个表达式都包含用 and 联结的 3 个比较运算式，可能要经过 9 次比较运算才能得出最大值。对于 n 很大的情形，这个算法最坏需要(n-1)2 次比较才能得到结果，效率很差，另外在代码形式上也会很难看（用 and 联结起来的 n-1 个比较运算式的长度远远超出了屏幕上一行的宽度）。

上述算法的问题在于：对每个数据的检测是独立设计的，一个数据的测试信息不会被后 面的测试利用。例如，假设第一个分支发现 x1 大于 x2 但小于 x3，这时我们能够推知 x3 是 最大值。但是上述代码却完全忽略这个信息，只是进入第二个分支继续检测，直至到第三个 分支才得出 x3 是最大值。

策略 2：判定树

执行比较运算 a>b 后，也许不能得出最大值是哪个数据，但肯定可以推知某个数据不是 最大值。因为若 a 大于 b，则 b 不可能是最大值；否则 a 不可能是最大值。后续的比较测试 可以充分利用这个信息，以避免冗余测试。根据这个思路，我们可以将所有测试安排一个合 理的顺序，以便排在后面的测试能够利用前面测试的信息。判定树方法就是这么一种安排测 试顺序的常用方法。假设我们从测试 x1>=x2 开始，如果这个比较运算结果为真，那么接下 去只需要测试 x1 与 x3 的大小，否则只需要比较 x2 和 x3 的大小。可见，每一次测试都产生 两个分支，每个分支又是一次测试，又产生两个分支。如此继续下去，最终形成一个层次结 构，称为判定树（见图 3.12）。

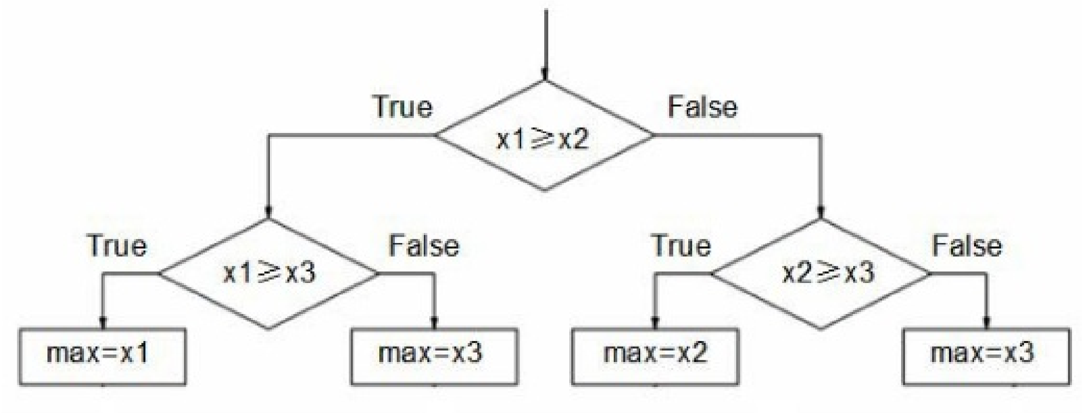


图 3.12 判定树

我们很容易根据判定树作出程序的流程图，并进而转化成 if-else 语句：

```
if x1 &gt;= x2:
    if x1 &gt;= x3:
        max = x1
    else:
        max = x3
else:
    if x2 &gt;= x3:
        max = x2
    else:
        max = x3
```

分析一下图 3.12 中的判定树（或者分析上面的 if 语句也一样）即可发现，为了求得最大 值，只需沿着自顶向下的某一条测试路径走到底即可，而任一路径上的比较运算次数都是两 次。所以，不管三个数值的大小次序是什么，上述算法都只进行两次比较运算，就能得出最 大值。效率与第一种策略要高。但是，这个方法导致的代码结构更加复杂，仍然不适合处理 较大的 n。例如，如果是求 4 个数据中的最大值，就会导致 3 重嵌套的 if-else 语句。

策略 3：顺序处理

前面两种策略都不适合对很多数据求最大值。还有更好的方法吗？

在为一个问题设计算法时，建议读者可以先问问自己：如果是你，你会如何解决该问题。 就此例而言，对于找三个数的最大值问题，你可能不会费脑筋多想，因为只需看看三个数值 就知道最大值了。但是如果交给你一本数据记录，其中有成千上万的数据，而且没有特定顺序，你又会怎么找出其中的最大值呢？

相信你一定会想出这个简单的策略：从头到尾逐一检查每个数值，心中记住当前见过的 最大值；每当遇到更大的数值，就用它替换心中所记的数值。这样，等到所有数据都检查过 了，最后记在心里的就是最大值。

将这个策略写成计算机算法，只需用一个变量（用 max 就好）来记录当前见过的最大值。 当处理完所有数据，max 中存放的就是全体数据中的最大值。下面的代码是三个数据的版本：

```
max = x1
if x2 > max:
    max = x2
if x3 > max:
    max = x3
```

分析一下这个顺序处理策略可知，它只需要进行两次比较运算就能得到最大值，这一点和第二种策略一样。但是顺序处理策略的代码比第二种策略简单得多，不需要嵌套的 if 语句。 更重要的是，这个策略是可扩展的，能够推广到任意 n 个数据的情形而不降低效率。例如， 如果有 4 个数据，我们只需增加一行语句：

```
max = x1
if x2 > max:
    max = x2
if x3 > max:
    max = x3
if x4 > max:
    max = x4
```

或者更简洁地用一个循环来表示，那样连数据变量也可以公用，无需使用 4 个独立变量。 将上述算法推广到对任意 n 个数据求最大值的情形，即可得到一般的求最大值的程序。

代码如下：

【程序 3.12】maxn.py

```
n = input("How many numbers? ")
max = input("Input a number: ")
for i in range(n-1):
    x = input("Input a number: ")
    if x &gt; max:
        max = x
print "max =", max
```

不难看出，为了从 n 个数据中求得最大值，这个程序只需要执行 n-1 次比较运算。

策略 4：利用现成代码

最后值得一提的是，Python 其实有一个内建函数 max，其功能就是返回若干个数据中的 最大值。如果使用这个函数，代码就简单到了极致，在交互环境下就能方便地解决问题：

```
>>> x1,x2,x3 = input("Input three numbers: ")
>>> print "max =", max(x1,x2,x3)
```

当然，这简直已称不上是一个算法，对我们学习程序设计没什么帮助。

3.6.2 经验总结

求最大值问题并非很难的问题，但解决该问题的过程反映了一些有关算法和程序设计的 重要的思想。

对于一个比较复杂的计算问题，往往有多种解决方法。作为算法设计者，通常不要凭着 第一感去编写代码，而是应当三思而后行。即使已经设计出了一个算法，也应当多问自己是 否还有更好的解法。

程序设计的首要任务是找到正确的算法，然后就应当去追求清晰的程序结构、代码的执 行效率、功能的可扩展性、良好的风格等目标。好的算法和程序就像逻辑的诗歌。读和维护 都很愉快，

虽然我们编写的程序是让计算机执行的，但在设计解决问题的算法时，常常可以站在人 类的立场考虑，问问自己假如是人类去解决这个问题，会有什么好方法？人类在生活实践中 积累了大量的行之有效的思考方式和解决办法，它们往往可以应用到计算机程序设计当中。 案例中虽然我们考虑的是三个数据求最大值的问题，但在设计过程中我们的思路并不局 限于这个特例问题。事实上，我们时时会考虑某个解决方法是否适用于更一般的 n 个数据的 情形。计算机程序设计中常有这种情形，通过考虑一般问题得到的算法，往往比只考虑特例 问题得到的算法还要好。因此，在设计程序时应该多考虑如何使程序更一般化，毕竟一般化的程序有可能应用到更多的问题当中。

上一节的第四个策略利用了现成的 max 函数，不能认为这是程序设计中的“投机取巧”。 相反，这个例子反映了一条重要经验：很多聪明的程序员已经设计出了无数好的算法和程序。当你所要解决的问题看起来很普通，可能有很多人已经遇到过这个问题，那么你就可以试着寻找该问题的现成解法。初学编程时可以尽量自己从头开始设计一个算法，但职业程序员都懂得借鉴、重用代码。

3.7 Python 布尔表达式用作控制结构*

有了顺序、分支和循环控制结构，原则上已足以表达所有算法。然而，为了在解决某些问题时编程更加方便，各种语言还提供了若干其他控制结构。本节介绍 Python 的一个特色， 即布尔表达式可当作控制结构来用。

编程语言中的表达式本来只是用来产生值的，布尔表达式也不例外。布尔表达式的常规 用法是计算产生 True 或 False，并用在分支和循环控制结构当中。但 Python 中的布尔表达式 还可以用作控制结构，这是由 Python 在底层计算布尔表达式时所采用的计算策略决定的。为了理解布尔表达式如何用作控制结构，需要了解 Python 是如何实现布尔运算的，详情见第 2 章。

考虑用一个交互式循环来实现“yes or no”功能：程序询问用户一个问题，用户输入回 答。只要用户输入的字符串以“y”或者"Y"开头，就算该用户回答是 yes，程序再进行合适的 处理；否则就跳过处理过程。这个功能很容易用 while 循环语句实现：

```
answer = raw_input("Want to play?(yes or no) ")
while answer[0] == "y" or answer[0] == "Y":
    play()
    answer = raw_input("Want to play?(yes or no) ")
```

显然这里 while 语句中的条件表达式等同于自然语言中的“用户输入以 y 打头或者用户 输入以 Y 打头”。然而，自然语言一般不会这么罗嗦，更简洁的表达是“用户输入以 y 或 Y 打头”。可惜这种简明的表达在编程语言中常常是错误的，初学编程者受自然语言的影响，很 容易写出下面这样的布尔表达式：

```
while answer[0] == "y" or "Y":
```

上面这个布尔表达式的写法在大多数语言中都导致语法错误，因此能够被编译器或解释器发现，不会造成严重后果。但是在 Python 中，这个表达式的语法却完全没有问题。然而，它的语义却很有问题，事实上，这个布尔表达式会导致一个无穷循环！原因就在于 Python 在底层实现布尔运算时所采取的“捷径”策略。

我们来看表达式 answer[0] == "y" or "Y"的计算。布尔运算符 or 所连接的两个表达式分别 是 answer[0] == "y"和"Y"，左边的表达式是真正的布尔表达式，计算结果为 True 或 False；而右边的表达式是一个字符串，它的值就是固定的非空串"Y"。根据第 2 章中介绍的 Python 对运算符 or 的计算规则：若 answer[0] == "y"计算到 True，则整个布尔表达式的值就是 True，不去考虑右边的表达式；若 answer[0] == "y"计算到 False，则整个布尔表达式返回值"Y"，这个非空串被 Python 视为 True。总之，不管用户输入的是什么，表达式 answer[0] == "y" or "Y" 永远为真，亦即 while 循环是无穷循环。

Python 的这个特性对初学者来说是个潜在的陷阱，很容易犯错误。当然，Python 之所以 如此设计，也有它的理由，那就是布尔表达式可以用作控制结构，在某些情况下可以写出更 简明的代码。例如考虑这种需求：程序要求用户输入一个字符串，如果用户没有输入数据就 直接按了回车键，则程序采用缺省值"Python"。实现这种需求的代码如下：

```
ans = raw_input("What's your favorite? [Python] ")
if s != "":
    favorite = ans
else:
    favorite = "Python"
```

利用字符串可被 Python 解释为布尔值的特性，上面代码中 if 语句的条件可以简化成：

```
ans = raw_input("What's your favorite? [Python] ")
if ans:
    favorite = ans
else:
    favorite = "Python"
```

当用户直接按回车，则 ans 为空串，并被 Python 解释为 False，从而 favorite 被赋值为缺省值"Python"。再利用布尔运算 or 的计算捷径规则，代码可以进一步简化为：

```
ans = raw_input("What's your favorite? [Python] ")
favorite = ans or "Python"
```

根据 or 的计算规则，此处第二行语句中的 ans or "Python"等同于一个 if-else 结构，即：若 ans 非空，就直接返回它的值；若 ans 为空串，则返回"Python"。这就是我们所说的“布尔表达式 用作流程控制结构”。

顺便说一下，如果考虑到 ans 其实就是函数 raw_input 的返回值，这个例子最终可以精简 成一行代码：

```
favorite = raw_input("What's your favorite? [Python] ") or "Python"
```

与上面第一个版本（5 行代码）相比，显然代码量大大减少了。看上去似乎不错，但其实程序的可读性也大大降低了，因为最后这个一行语句的版本对初学者来说是很难理解的。从良好编程风格的角度说，宁可多写几条语句，也要保证程序的可读性和以理解性。

3.8 练习

1. 程序流程的基本控制结构有哪几种？

2. 单分支、两路分支和多路分支的 if 结构分别是怎样的？

3. 传统的错误检测代码是怎样的？

4. 现代编程语言为什么引入异常处理机制？Python 的 try-except 语句的用法是怎样的？

5. for 循环结构有哪几种用法？

5. while 循环结构有哪几种用法？

6. 如何将 for 循环结构转化为 while 循环结构？

7. 结构化程序设计的基本内容有哪些？

8. try-except 语句、break 语句、continue 语句是否合乎结构化程序设计的原则？

9. 好的程序具有哪些特征？

10. 设计程序：输入一个数值，输出该数值是正数、负数还是 0 的信息。

11. 设计程序：输入体重（公斤）、身高（米），计算身体质量指数 BMI，并输出健康信息。提示：BMI＝体重／身高的平方。BMI 在 19 以下为轻体重，[19,25)之间为健康体重，[25,28)为超重，28 以上为肥胖。

12. 设计程序：输入百分制的考试分数，输出相应的等级制名称。设 A：90－100，B：80－89，C：70－79，D：60－69，F：59 以下。

13. 设计程序：输入年份，输出该年是否闰年。提示：如果年份能被 4 整除，并且当它能被 100 整除的时候也能被 400 整除，则该年是闰年。

14. 设计程序：输入三个数据，分别代表操作码（'A'、'S'、'M'、'D'，分别表示加、减、乘、除）和两个操作数，输出操作数按操作码进行计算后的结果。

15. 设计程序：计算 Fibonacci 数列的第一个大于 100 的数。

16. 设计程序：输入 n，输出 11 + 22 + 33 + ... + nn 。

17. 设计程序：用 1 元钱买价格小于 1 元的物品，用 1 分、2 分、5 分、1 角、2 角和 5 角的硬币找零，要求找回的硬币数量最少。

18. 设计程序：输入考试分数求和。要求第一个输入是数据个数，其他输入是分数；只有超过 60 的分数才求和；累计及格分数的个数；最后输出总分和及格分数的个数。

19. 设计程序：计算从 1 到 1000 的能被 3 整除且不能被 5 整除的所有整数之和。

20. 设计程序：输入自然数 m 和 n，输出 m 和 n 之间所有奇数的和。要求能多次输入并计算。

21. 设计程序：利用p/4 = 1 – 1/3 + 1/5 – 1/7 + ... 求p的近似值。要求一直计算到所用的最后两 项的差小于 0.00001。提示：通项公式为(–1)n / (2n–1)。

第 4 章 模块化编程

随着待解决的问题越来越复杂，程序也越来越复杂。对于复杂问题，如果仅仅依靠上一 章介绍的结构化编程方法，是很难驾驭程序的复杂性的。因为在控制结构这个层次上考虑程 序设计，必然因两方面的复杂性而导致编程困难：一是在广度上有成千上万行的代码，二是 在深度上有多层嵌套的控制结构。为了简化复杂程序在代码形式上的复杂性，以便在较高抽 象层次上把握复杂程序，计算机科学家提出了模块化编程方法。

4.1 模块化编程基本概念

4.1.1 模块化设计概述

模块化设计的思想在许多行业中早已有之，并非计算机科学所独创。

例如，建筑行业很早就提出了模块化建筑概念，即在工厂里预制各种房屋模块构件，然 后运到项目现场组装成各种房屋。模块构件在工厂中预制，便于组织生产、提高效率、节省 材料、受环境影响小。模块组装时施工简便快速、灵活多样、清洁环保，盖房子就像儿童搭 建积木玩具一样。①

再如，船舶工业广泛采用模块化造船方法，即对最终产品（整艘船舶）进行层次化分解， 并以中间产品（部件、分段、总段等）作为生产单元，最后再逐级组装成为最终产品。模块 化设计和建造能够使问题简化、结构优化、功能单元化、目标多样化，能够降低成本、缩短 周期，使产品易于维护、更新和系列化。

又如，现代电子产品功能越来越复杂、规模越来越大，利用模块化设计的功能分解和组 合思想，可以选用模块化元件（如集成电路模块），利用其标准化的接口，搭建具有复杂功 能的电子系统。模块化设计不但能加快开发周期，而且经过测试的模块化元件也使得电子系 统的可靠性大大提高，标准化、通用化的元件使得系统易构建、易维护。

总之，模块化设计和建造就是在对产品进行功能分析的基础上，将产品分解成若干个功 能模块，预制好的模块再进行组装，形成最终产品。这里，模块（module）是指提供特定功 能的相对独立的单元。模块一般具有如下特征：

- 标准化：模块是具有标准尺寸和标准接口的预制功能单元，这是组装、互换等特征 的基础。
- 可组装：多个模块可以方便、灵活地组合、配置，以构造不同大小、不同形状、不 同功能的系统。
- 可替换：通过用一个模块去更换另一个模块，可以改变系统的局部功能而不影响系 统的其他部分。
- 可维护：可以对模块进行局部修改或设置，以满足用户的需求。另外可以在现有系 统中增加新模块，以扩展系统功能。

模块化概念最早应用于工程技术领域，针对的是物理产品，后来逐渐演变成更广义的概 念，并在许多非物理产品领域中得到应用。尤其是在本书讨论的程序设计领域，模块概念和 模块化设计方法已经成为广泛采用的方法。

4.1.2 模块化编程

模块化编程（modular programming）是一种软件设计技术，它将软件分解为若干独立 的、可替换的、具有预定功能的模块，每个模块实现一个功能，各模块通过接口（输入输出 部分）组合在一起，形成最终程序。

> ① 远大公司在模块化建筑领域的两个案例：6 天建成 15 层宾馆，15 天建成 30 层的 T30 酒店。

对于简单问题，可以直接构建单一模块的程序。而对于复杂问题，则可以先创建若干个较小的模块，然后将它们组装、链接在一起，从而构成复杂的软件系统。模块化编程具有以 下优点：

- 易设计：较大的复杂问题分解为若干较小的简单问题，使我们可以从抽象的模块功 能角度而非具体的实现角度去理解软件系统，从而整个系统的结构非常清晰、容易 理解，设计人员在设计之初可以更加关注系统的顶层逻辑而非底层细节。
- 易实现：模块化设计适合团队开发，因为每个团队成员不需要了解系统全貌，只需 关注所分配的小任务。另外团队可以灵活地增加人手，新人只需直接接手某个模块， 不会影响系统其他模块的开发。
- 易测试：每个模块不但可以独立开发，也可以独立测试，最后组装时再进行联合测 试。
- 易维护：如果需要修改系统或者扩展系统功能，只需针对特定模块进行修改或者添 加新模块。
- 可重用：很多模块的代码都可以不加修改地用于其他程序的开发。

模块化编程实际上是一条抽象设计原则的具体体现，即分离关注点（Separation of Concerns，缩写为 SoC）原则。所谓关注点，是指设计者关心的某个系统特性或行为；而分离关注点是指将系统分解为互不重叠的若干单元，每个单元对应于一个关注点。在模块化编程中，以程序的各个功能作为关注点，模块划分就是分离关注点的结果。一个模块可以使用另一个模块来实现自己的功能，但除此之外模块之间最好没有交互，这是 SoC 原则的理想 目标。

4.1.3 编程语言对模块化编程的支持

在 1950 年代，由于计算机内存容量很小，程序员们千方百计地想尽量减小程序的大小。汇编语言中最早出现了子例程（subroutine）和宏（macro）的构造，其目的正是为了减小程序大小。子例程和宏可以实现了“一次编写、多处多次使用”，从而避免了在程序中的重复代码，缩短了代码长度。

从 1960 年代开始，高级编程语言中出现了支持模块化编程的语言构造，这种构造在不同语言中可能有不同的名称和形式，除了上面提到的子例程之外，还有子程序（subprogram）、过程（procedure）、函数（function）以及由过程和函数组成的模块、包（package）等构造。以下我们用“子程序”来泛指这些模块化构造。

子程序是指程序中的一段代码，它执行特定任务，并且与同一程序中的其他部分是相对独立的。顾名思义，子程序也是程序，也是由许多计算步骤构成的指令序列；但抽象地看，可以将一个子程序视为一个操作或高级指令，可以作为更大的程序中的一个简单步骤来使用。在程序的一次执行中，可以多次、多处执行子程序。子程序概念虽然仍然有避免重复代码、减小程序大小的作用，但其更重要的目的是使程序更加模块化。

子程序构造一般涉及以下一些内容：

- 子程序的创建：定义子程序的名字和代码（程序体）。
- 子程序的调用和返回：调用就是要求执行子程序，而子程序执行完毕应当将控制返回给调用者。
- 参数：相当于子程序所需的输入数据，一般需要预先声明参数的类型和个数，并在调用时提供具体的参数值。
- 返回值：相当于子程序的输出数据，一般需要预先声明返回值的类型。
- 局部变量：子程序中定义的变量在子程序外部是不可见的，亦即子程序构成了一个私有名字空间。这是子程序独立性的一种表现。
- 全局变量：子程序外部定义的变量如果被声明为全局变量，那么所有子程序都可以共享使用、操作该变量。

有的编程语言（如 Pascal 语言）同时提供两种子程序构造：过程和函数。过程不产生返回值，因此总体上相当于一条命令语句，只规定了要执行的操作；而函数不但要执行一些计算，更重要的是需要将计算结果返回给调用者，因此函数在使用时相当于一个数据。

有的编程语言（如 C 语言）则不将子程序区分为过程和函数，而是统称为函数。过程就是没有返回值的函数。不过，更现代的语言（如 Python）要求函数必须具有返回值，“过程”其实是返回某种特殊值（如 Python 中的 None）的函数。

子程序是传统的过程式语言中的模块化构造。模块化编程方法经过多年发展，又派生出 了面向对象编程方法。在面向对象语言中，对象实际上就是模块概念的推广，传统模块之间 的调用接口相应地发展成了对象之间的消息传递接口。

一个编程语言一般只提供基本的编程构造（数据类型、语句、子程序等），用户所需的 实用功能都必须由自己编程实现。为了帮助用户进行应用开发，专业软件开发商一般会为某 种编程语言开发很多提供标准功能的子程序，用这种语言编程时可以直接调用这些标准子程 序。这些标准子程序构成了所谓的程序库，它是软件重用、共享和营销的重要形式。例如 math 和 string 就是 Python 语言的标准库。同样地，面向对象编程语言则以类库的形式 为程序员提供大量的标准功能代码。

总之，子程序是强大的模块化编程工具，通过将复杂程序分解为子程序，可以大大降低 开发复杂程序的难度，使问题变得可理解、易开发。另外，子程序的独立性还意味着可以由 团队来开发复杂程序，从而提高软件生产率。最后，由于较小的子程序更容易验证正确性， 所以模块化开发还可以保证复杂程序的质量和可靠性。

4.2 Python 语言中的函数

在数学中，函数是一种映射，其功能是将自变量的值（输入）映射到一个函数值（输出）。

编程语言中的函数是一段程序代码，其功能是根据输入（参数）进行计算，并产生输出（返回值）。从上一节我们了解了模块化编程的一般知识，并且知道函数是一种常见的子程序构造，是模块化编程的基本工具。对于 Python 语言，函数是最重要的语言构造之一，本节具体介绍 Python 语言中的函数。

从前面几章，我们已经见过 Python 的一些内建函数（如 abs、len 等）、Python 标准库 中的函数（如 math.sqrt、string.split 等）， 下一章我们还会看到对象的方法也是一 种函数。本节要讨论的函数是用户自定义函数。

编程语言中为什么要引进用户自定义函数这种构造呢？

4.2.1 用函数减少重复代码 首先看一个简单的用字符画一棵树的程序：

【程序 4.1】tree1.py

```
print "   *   "
print "  ***  "
print " ***** "
print "*******"
print "   *   "
print "  ***  "
print " ***** "
print "*******"
print "   #   "
print "   #   "
print "   #   "
```

执行结果如下：

```
   *
  ***
 *****
*******
   *
  ***
 *****
*******
   #
   #
   #
```

尽管程序 4.1 实现了我们预定的功能，但从程序的形式、风格角度看，还是有不足之处。 从程序可见，代码的 1～4 行和 5～8 行是完全相同的①，它们对应于树冠的上下两部分。一 个程序中如果多处出现相同代码，会带来三个问题：第一，重复输入相同代码很烦人；第二，重复代码使程序不必要地增加长度；第三，也是最重要的一点，代码维护很麻烦。前两条很容易理解，我们来说明一下第三点。代码维护是指修改代码等工作。当要修改的代码在多处重复出现时，显然必须在每一个重复出现处做统一的修改，以保持重复代码的一致性，这就增加了代码维护的难度。

对程序 4.1 来说，重复代码很少，不算什么大问题。然而，如果重复代码很长、重复次 数很多，上述三个问题就不是可以忽视的了。事实上，多次键入重复代码至少会增加输入出 错的可能性，而维护重复代码时也很容易忘记在各处统一修改，这些都会导致重复代码的不 一致。至于重复代码使程序拖沓冗长，就更不必说了。

如何解决这种重复代码问题呢？函数正是我们所需的语言构造。 我们已经知道，函数是一个子程序，其基本思想是将一个语句序列看作一个整体，并为

该语句系列命名。此后，在程序中的任何地方，只要引用该函数名，就能执行函数的语句序列。创建函数的代码称为函数定义，以后使用函数的代码称为函数调用。

下面我们定义一个函数 treetop()，它的语句序列正是程序 4.1 中的重复代码。注意， 为了更直观地介绍函数定义及其调用，我们特意在 Python 交互环境 IDLE 中来展示有关内 容。

```
>>> def treetop():
    print "   *   "
    print "  ***  "
    print " ***** "
    print "*******"
```

def 语句只是定义了新函数 treetop，并没有执行函数体中的语句，因此不会产生显 示输出。直到调用 treetop 函数时，才执行函数体。我们来看看它的功能是什么。

> ① 如果读者自己在文本编辑器中键入这个程序，一定会使用“复制－粘贴”功能吧。

```
>>> treetop()
   *
  ***
 *****
*******
```

可见函数 treetop 正确地打印了树冠的一部分。 接下来定义画出整棵树的函数 tree：

```
>>> def tree():
    treetop()
    treetop()
    print "   #"
    print "   #"
    print "   #"
```

由于重复代码被函数调用 treetop 代替，这个版本显然比原先的版本简练许多，但程序的 功能完全是一样的，参见下面的运行结果：

```
>>> tree()
   *
  ***
 *****
*******
   *
  ***
 *****
*******
   #
   #
   #
```

至此我们用函数解决了重复代码的问题。要注意的是，我们是在交互环境下展示函数定 义和调用的，因而可以先定义函数 treetop 并单独运行此函数，然后再定义主函数 tree 并运行之。如果按通常的做法将代码保存为程序文件，则应将两个函数合并为一个程序文件 来保存，因为它们不过是一个程序的两个部分而已。即如程序 4.2 所示。

【程序 4.2】tree2.py

```
def treetop():
    print "   *   "
    print "  ***  "
    print " ***** "
    print "*******"
def tree():
    treetop()
    treetop()
    print "   #"
    print "   #"
    print "   #"
tree()
```

顺便说明一下，程序 4.2 中定义了两个函数，其中 tree 是主函数，用于完成程序的总 体功能，而 treetop 是辅助性的函数（子程序），用于完成部分功能。其中最后一行是调 用主函数，这是启动整个程序的入口。作为惯例，我们通常将一个程序的主函数（程序入口） 命名为 main。今后，我们给出的例子程序即使并未定义辅助性的函数，我们也将所有代码 置于一个主函数 main 之中，这是惯例，也符合模块化编程的风格——程序至少由一个主控 模块构成。

有的读者也许会问，程序 4.2 中的函数 tree 中，还存在三条重复出现的语句

```
print "   #"
```

为何不定义一个函数来避免重复呢？我们不妨再写一个新版本，读者看了之后自然明白这个做法没什么好处。见下：

```
def treetop():
    print "   *"
    print "  ***"
    print " *****"
    print "*******"
def printhash():
    print "   #"
def tree():
    treetop()
    treetop()
    printhash()
    printhash()
    printhash()
tree()
```

从这个版本可以看出，由于重复的代码只是一条语句，如果为重复代码定义一个新函数， 不但不能使代码精简，反而使代码变复杂了。更重要的是，利用函数来取代重复代码不是没 有代价的，因为函数调用和返回都需要花费系统开销。这个版本花了代价，却没有带来任何 收益，所以是不合适的。

4.2.2 用函数改善程序结构

上一节讨论了函数的减少重复代码、精简程序的作用，并利用函数的这个功能将程序 4.1 改进成了程序 4.2。在该节的最后，我们也给出了一个不宜用函数来减少重复代码的情况。

还能不能利用函数将程序 4.2 变得更好呢？

我们在 4.1 节中一般地讨论了模块化编程，在 Python 中，函数就是用于模块化编程的 重要工具。当算法很复杂时，程序就会变得难以理解。据说人类擅长同时应付 8 到 10 件事 情，当面对成百上千行的算法时，最好的程序员也会感到难以把握。应对程序复杂性的一种 方法就是模块化，将程序分解成多个较小的相对独立的子程序。下面我们来看程序 4.2 还能 怎样改进。

我们定义一个新函数 treetrunk，它的语句序列就是程序 4.2 的主函数中用于画树干的三条 print 语句。即：

```
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
```

然后我们用这个函数取代主函数的那三条 print 语句，就得到画树程序的一个新版本。

【程序 4.3】tree3.py

```
def treetop():
    print "   *"
    print "  ***"
    print " *****"
    print "*******"
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
def main():
    treetop()
    treetop()
    treetrunk()
main()
```

注意我们将程序主函数的名字从 tree 改成了更符合惯例的 main。

简单地比较一下程序 4.2 与 4.3 这两个版本就看出，由于多了函数 treetrunk 的定义与调用，新版本的代码不但没有减少，反而增加了。那为何要引进 treetrunk 函数呢？其实我们的目的是使主程序的结构更清晰，从而更容易理解程序功能。通过将一些实现细节转移到一个单独的函数中，并对函数进行合适的命名，能够使程序的可读性大大增强。例如我 们来读程序 4.3 的主程序 main，就会发现该程序不过是先画树冠（由两个相同形状组成）， 再画树干而已，程序的功能一目了然。

如果进一步发挥上述思想，就会发现程序 4.3 的结构还不够完美。问题出在主程序的第 一步——画树冠，这一项任务逻辑上是个整体却用了两个函数调用来完成，这就好比老师对 学生说“请大家画上一半树冠，再画下一半树冠”，显然不如直接说“请大家画树冠”来得 清晰易懂。因此，我们再引入一个新函数用于隐藏树冠的实现细节（上下两部分），从而得 到程序 4.4，这个版本在避免重复代码和模块化两方面可以说达到了完美。

【程序 4.4】tree4.py

```
def treetop1():
    print "   *"
    print "  ***"
    print " *****"
    print "*******"
def treetop():
    treetop1()
    treetop1()
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
def main():
    treetop()
    treetrunk()
main()
```

现在再来读主程序 main，显然更容易理解了——从程序顶层看，整个程序不外乎就是画树冠、画树干两步而已。如果只想了解程序的总体功能，那么读懂 main 函数就够了；如果还想了解更多细节，那就再去读辅助函数 treetop1 和 treetrunk 等。

读者在编程时应当多模仿、多体会程序 4.4 中函数的用法，并学会欣赏模块化程序在结 构方面的优美。

4.2.3 用函数增强程序的通用性

我们说过，程序 4.4 在减少重复代码和模块化两方面已经做得很好，但这并不意味着该 程序在各方面都已经完美。例如，如果我们希望换用字符"^"再画一棵树，以便比较哪个更 好看些，该如何做呢？显见的做法是仿照用"*"画树的代码重写画树冠的函数，而树干部分 可以重用。于是得到下面的代码：

【程序 4.5】tree5.py

```
def treetop1():
    print "   *"
    print "  ***"
    print " *****"
    print "*******"
def treetop2():
    print "   ^"
    print "  ^^^"
    print " ^^^^^"
    print "^^^^^^^"
def star_treetop():
    treetop1()
    treetop1()
def caret_treetop():
    treetop2()
    treetop2()
def treetrunk():
    print "   #"
    print "   #"
    print "   #"
def main():
    star_treetop()
    treetrunk()
    print
    caret_treetop()
    treetrunk()
main()
```

此版本的执行结果如下：

```
    *
   ***
  *****
 *******
    *
   ***
  *****
 *******
    #
    #
    #
    ^
   ^^^
  ^^^^^
 ^^^^^^^
    ^
   ^^^
  ^^^^^
 ^^^^^^^
    #
    #
    #
```

虽然程序 4.5 满足了功能需求，但是从程序设计角度说是很笨拙的，因为这是一种“头 痛医头脚痛医脚”的方法，即为每一种特殊情形创建新的代码。更好的做法是用一个一般的 函数来处理所有特殊情形。鉴于 treetop1 和 treetop2 的非常类似，我们可以从他们抽 象出一个通用的画树冠的函数，使得该函数能够取代 treetop1 和 treetop2。

函数的通用性可以通过引入参数（parameter）来实现。要理解参数的作用，可以简单地与数学函数的自变量进行类比。以函数 f(x) = x2 为例，对于给定的自变量值 10，函数计算 出函数值 f(10) = 100；换不同的自变量值 20，则函数又计算出另一个函数值 f(20) = 400。编 程语言中的函数参数具有类似的行为，输入不同的参数值，则函数执行后可产生不同的结果。

下面我们设计一个通用的画树冠的函数 treetop(ch)，其中参数 ch 表示用来作画的 字符。为了控制树的形状，函数定义中使用了字符串格式化运算。

```
>>> def treetop(ch):
        print "   %s" % (ch)
        print "  %s" % (3 * ch)
        print " %s" % (5 * ch)
        print "%s" % (7 * ch)
```

在交互环境定义了函数 treetop(ch)后，我们接着来测试它的效果。下面是测试例子：

```
>>> treetop('*')
   *
  ***
 *****
*******
>>> treetop('^')
   ^
  ^^^
 ^^^^^
^^^^^^^
>>> treetop('A')
   A
  AAA
 AAAAA
AAAAAAA
```

可见函数 treetop(ch)确实具有通用性，只要为它的参数提供一个字符值，就能用该字符 画出树冠形状。下面我们利用 treetop(ch)函数来改写程序 4.5：

【程序 4.6】tree6.py

```
def treetop(ch):
    print " %s" % (ch)
    print " %s" % (3 * ch)
    print " %s" % (5 * ch)
    print "%s" % (7 * ch)
def star_treetop():
    treetop("*")
    treetop("*")
def caret_treetop():
    treetop("^")
    treetop("^")
def treetrunk():
    print " #"
    print " #"
    print " #"
def main():
    star_treetop()
    treetrunk()
    print
    caret_treetop()
    treetrunk()
main()
```

此版本的执行结果和程序 4.5 完全一样，但是比较两者的代码会发现，程序 4.6 将程序 4.5 中的两个函数合二为一，增强了通用性。以后如果想换用其他字符画树，修改程序 4.6 比修改程序 4.5 要简单得多。

4.2.4 小结：函数的定义与调用

通过前面的例子，读者应该已经非常熟悉 Python 中函数定义的语法。在此总结如下：

```
def <函数名>(<形式参数>):
    <函数体>
```

其中函数名是标识符，命名必须符合 Python 标识符的规定；形式参数是用逗号分隔的变量名序列（可以为空）。函数体是语句序列，左端必须缩进一些空白。 一旦定义了一个函数，就可以在程序的任何地方调用这个函数。函数调用的语法如下：

```
<函数名>(<实际参数>)
```

其中实际参数可以是表达式，个数必须和形式参数相同。注意，这里列出的函数调用语法实际上适用于没有返回值的函数，即 4.1.3 节中提到的“过程”。4.2.6 小节会讨论具有返回值 的函数。

当 Python 遇到一个函数调用时，将通过四个步骤来处理这个调用。假设程序 P 现在执 行到了一个函数调用 f(a)，则这四个步骤是：

（1）调用者 P 在调用点暂停执行（术语也称为 P 挂起）；

（2）函数 f 的形式参数被赋予实际参数 a 的值；

（3）执行 f 的函数体；

（4）f 执行完毕后，控制返回到 P 中调用点的下一条语句。

下面我们以程序 4.6 为例，具体描述函数调用过程。为了方便阅读，将程序 4.6 的主函 数 main 罗列在下面，整个程序从 main 开始执行。

```
def main():
    star_treetop()
    treetrunk()
    print
    caret_treetop()
    treetrunk()
```

当 Python 执行到 star_treetop()时，main 暂停执行，控制转到 star_treetop。 因为没有参数传递问题，所以直接执行 star_treetop 的函数体。图 4.1 描述了这个函数 调用的控制转移情况。

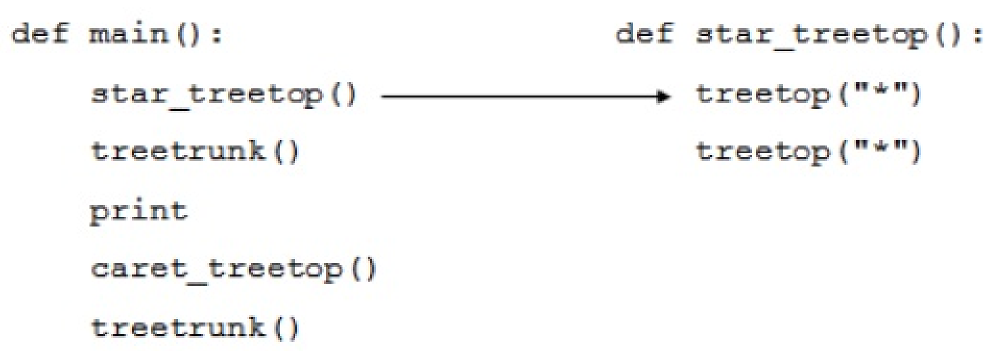


图 4.1 控制从 main 转移到 star_treetop

控制转到 star_treetop 后执行的第一条语句又是一个函数调用 treetop("")， 于是 *Python* 又暂停执行 *star_treetop*，而将控制转到 *treetop("")*。Python 检查 treetop 的定义后发现它有一个形式参数 ch，于是将函数调用 treetop("")的实际参 数""传递给形式参数 ch，这相当于在 treetop 的函数体之前增加了一条赋值语句：

```
ch = "*"
```

参数传递后开始执行 treetop 的函数体。图 4.2 展现了这时的状态，注意 treetop内部的变量 ch 已经被赋值为"*"。

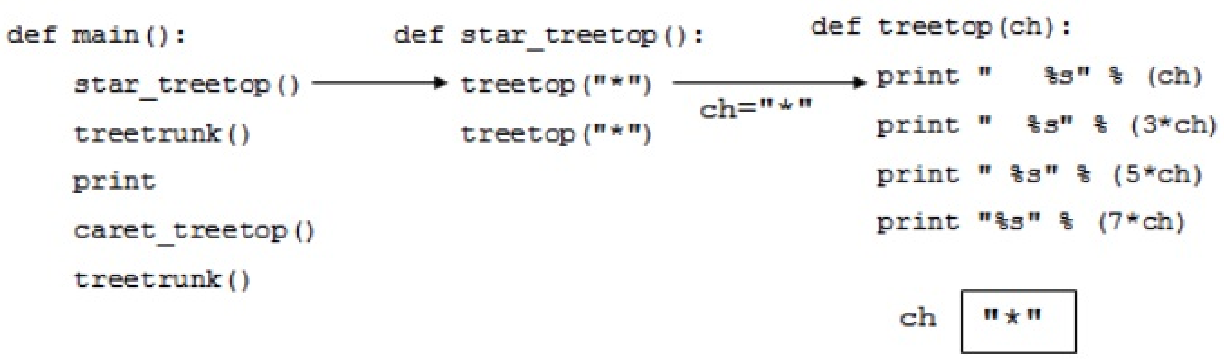


图 4.2 控制从 star_treetop 转移到 treetop

由于 treetop()的函数体是一系列 print 语句，没有更多函数调用，于是 Python 顺 序执行这些语句，结束后将控制返回到 treetop 调用点的下一条语句，即 star_treetop 中的第二条 treetop("*")语句，这时的情形参看图 4.3。注意，当函数执行完毕，函数的 变量所占用的存储空间将被 Python 收回，任何变量都不可能将数据保持到下一次执行函数， 故图 4.3 中 ch 显示为未赋值状态。

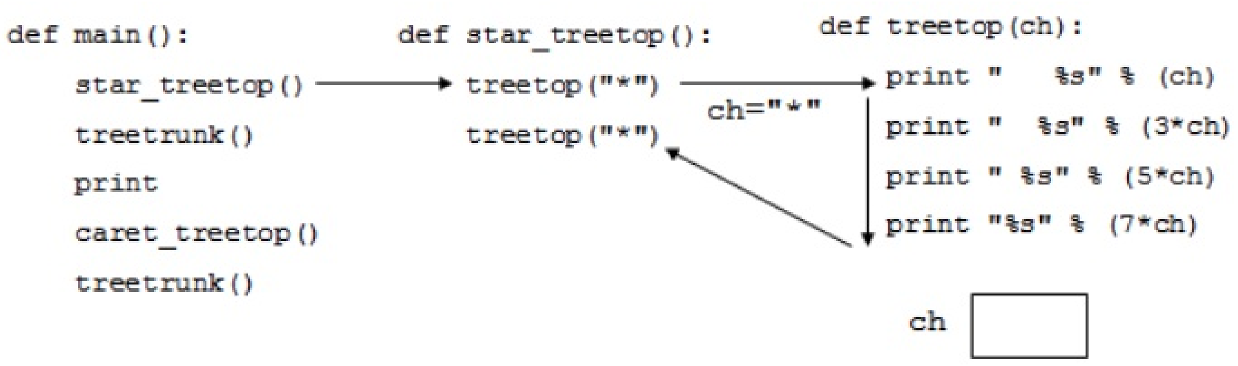


图 4.3 控制从 treetop 返回 star_treetop

接下来执行 star_treetop 的第二条 treetop("*")，其过程和前面一条完全一样， 我们就不作图演示了。现在，当控制从 treetop 再次返回 star_treetop 时，此函数也 执行完毕，故控制又返回到 main 函数中调用点的下一条语句。如图 4.4 所示。

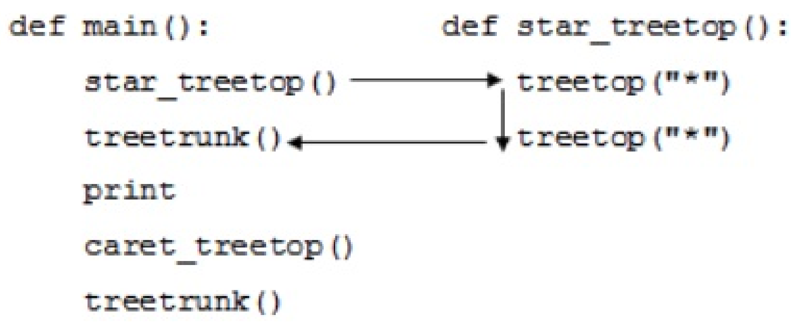


图 4.4 控制从 star_treetop 返回 main

控制返回 main 后执行的是第二条语句 treetrunk()，这又是一个函数调用。于是 main 再次暂停执行，控制转移到函数 treetrunk。treetrunk 执行完毕控制返回 main， 执行第三条语句 print，输出一个空行后执行函数调用 caret_treetop()。这和前面 star_treetop()的执行过程是类似的，控制转移到 caret_treetop 的函数体后遇到的 是 treetop("^")，这次传递给形式参数 ch 的值是字符"^"，图 4.5 表示了此时的状态。

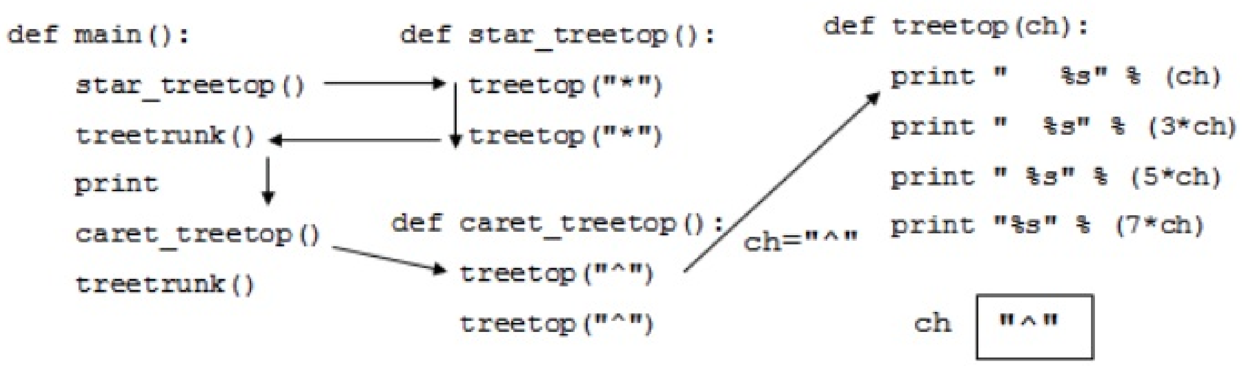


图 4.5 控制从 caret_treetop 转到 treetop 并传递不同实际参数

此后的执行过程与上述类似，我们不再逐一说明。当程序最后一行的调用 treetrunk 执行完毕，控制返回到 main 时到达程序末尾，于是整个程序结束。其实，main 本身也是一个函数，程序 4.6 的最后一行就是对 main 的调用。由于 main 是顶层模块，调用并执行 main 后控制只能返回给 Python——所以整个程序执行完毕后我们看到的是熟悉的“>>>”。

以上我们通过例子描述了 Python 的函数定义和调用。还要说明一点，函数定义中提到 形式参数可以是用逗号分隔的变量名序列。对于有多个形式参数的函数，调用时一定要注意 形式参数与实际参数的匹配。简单的做法是按位置匹配，即调用时提供的第一个实际参数赋 值给第一个形式参数，第二个实际参数赋值给第二个形式参数，依此类推。

作为例子，我们再来研究用字符画树冠的问题。树冠是由两个三角形图案组成的，程序 4.2 或程序 4.6 中，函数 treetop 的功能就是用字符画三角形图案，只不过程序 4.2 固定用 字符"*"画画，程序 *4.6* 可以用任意字符画画。观察 *treetop* 的函数体，可见图案是由多 条 *print* 语句所打印的字符串拼成的，并且每条 *print* 所打印的字符串很有规律：每行中 "*"的个数是自顶向下分别是 1、3、5、7，而左边留的空格数自顶向下分别是 3、2、1、0。 对这些数字做一点分析，很容易得出规律：设树冠最宽处有 w 个"*"字符，则当某一行上要 画 *c* 个"*"时，该行左边留的空格数就是(w - c) / 2。根据这个规律，我们定义一个新的 treetop 函数，它具有两个参数：一个是画图所用字符 ch，另一个是树冠宽度 width（为 对称起见应该用奇数，此前例子都固定为 7）。显然这个新的 treetop 函数更加通用化，可 以用任意字符画任意宽度的树冠。

```
def treetop(ch,width):
    for c in range(1,width+1,2):
        print ((width-c)/2) * " " + c * ch
```

下面我们在 Python 交互环境下定义这个函数，然后做一些测试。结果如下：

```
>>> treetop("*",7)
   *
  ***
 *****
*******
>>> treetop("@",9)
    @
   @@@
  @@@@@
 @@@@@@@
@@@@@@@@@
>>> treetop(11,"A")
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module> treetop(11,"A")
File "<pyshell#2>", line 2, in treetop for c in range(1,width+1,2):
TypeError: cannot concatenate 'str' and 'int' objects
```

从上例可知，由于函数 treetop 有两个形式参数，因此调用该函数时必须传递两个实 际参数与之匹配。参数传递的效果相当于在 treetop 的函数体前面执行了两条赋值语句：

```
ch = ...
width = ...
```

如果实际参数与形式参数不匹配，函数执行就可能出错，如上例中的 treetop(11,"A")。 更严重的是函数执行似乎没有出错，但参数的错误匹配实际上导致计算结果完全没有意义。

例如我们定义一个显示身高体重信息的函数，然后调用之：

```
>>> def printInfo(height,weight):
        print "Height:",height
        print "Weight:",weight
>>> printInfo(80,1.80)
Height: 80
Weight: 1.8
```

可见，由于调用时参数传递不匹配，函数虽然能够执行，但结果无意义。

关键字参数

函数调用时的参数传递通常采用上述“按位置匹配”的方式，但 Python 还提供另一种 参数传递方式——关键字参数。关键字参数形如“<形参名> = <实参值>”，即通过形式参数 的名字来指示为哪个形参传递什么值。例如：

```
>>> treetop(width = 11,ch = "A")
     A
    AAA
   AAAAA
  AAAAAAA
 AAAAAAAAA
AAAAAAAAAAA
```

关键字参数在某些场合用起来更方便。例如，如果一个函数有很多参数，但是调用时只 想为个别参数传递值，而其他参数采用缺省值，这是采用关键字参数就是必然的选择。下面 是一个简单的例子：

```
>>> def f(a,b=7,c=2):
        print a,b,c
>>> f(2005)
2005 7 2
>>> f(1927,8,1)
1927 8 1
>>> f(1921,c=1)
1921 7 1
```

注意，这个例子同时说明了如何为函数参数指定缺省值。

4.2.5 变量的作用域

程序中的变量都有自己的作用域（scope，或称辖域），即程序的一部分区域，在其中可以访问该变量。一个变量只有在它的作用域中才有定义，才能被访问。

局部变量

在一个函数中定义的变量称为局部变量（local variable），因为它们的作用域局限于该 函数的函数体，在函数外部是没有定义的。例如：

```
>>> def func(x,y):
        z = x + y print z
>>> func(1,2)
3
```

函数 func 中定义了局部变量 z。由于语句 print z 是 func 函数体内的语句，所以 可以访问 z。如果函数外部的 print 语句试图显示 z 的值，则会出错。例如接着上例继续 执行：

```
>>> print z
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module> print z
NameError: name 'z' is not defined
```

函数的形式参数也可以看作是函数的局部变量，即只能在函数体内访问。形式参数不同 于局部变量的是：形式参数的值是在调用函数时通过参数传递而来的。如上例中函数 func 有两个参数 x 和 y，当调用 func(1,2)时相当于执行了两个对局部变量的赋值语句 x = 1 和 y = 2。

函数的局部变量和形式参数仅在函数体内有定义，因此即使与函数外部的变量同名也不 会带来问题。例如我们接着上例继续执行语句：

```
>>> x = 1
>>> z = 2
>>> func(x,z)
3
```

这里，x 和 z 是在函数 func 的外部定义的变量，它们虽然分别与 func 的形式参数 x 和局 部变量 z 同名，但实际上毫无联系。执行 func(x,z)时，Python 先在 func 外部计算 x 和 z 的值，然后将结果传递给 func 的形式参数 x、y，因此最终执行的是 func(1,2)。图 4.6 给出了这个过程的示意图。

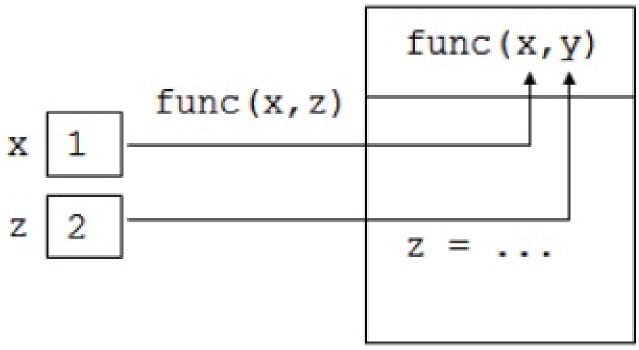


图 4.6 函数局部变量与外部变量同名 全局变量

函数内部的变量具有局部性，这符合模块化编程思想的要求。作为一种模块化构件，函 数就像“黑盒”一样，其内部细节应该对外部不可见。同理，函数内部也不应直接使用外界 的东西。如果函数需要外界的数据，正确的做法是通过参数来传递给函数。也就是说，函数 的参数除了用于表示可变数据、增强函数的通用性之外，还应作为外界向函数传递数据（即使是一个固定不变的数据）的唯一渠道。下面是一个函数直接使用外界数据的例子：

```
>>> s = "hello"
>>> def f():
        print s
>>> f()
hello
```

这里，函数 f()的功能是打印变量 s 的值，但这个 s 并不是 f()自己的局部变量，而是 f() 外部的变量，相对于 f()可称为全局变量（global variable）。尽管这个用法在 Python 中是合 法的，但这不是好的编程风格。正确的做法是将变量 s 的值通过参数传递给 f()：

```
>>> s = "hello"
>>> def f(x):
        print x
>>> f(s)
hello
```

在实际应用中，可能会有多个函数共同操作（读取或修改）一个数据的情形，这时采用 参数传递的方式比较麻烦，而采用全局变量则显得直接了当。下面我们用一个简单程序说明 Python 中全局变量的用法。

【程序 4.7】eg4_7.py

```
def f():
    global x
    x = x + 1
    print x
def g():
    global x
    x = x - 1
    print x
    x = 0
f()
g()
```

程序中定义了两个函数 f()和 g()，它们的函数体中都包含一条声明全局变量的语句：global x 意为本函数中所使用的 x 是在函数外部定义的全局变量。f()的功能是对全局变量 x 加 1，g()的功能是对全局变量 x 减 1。执行结果如下：

```
>>> import eg4_1
1
0
```

可见执行 f()之后 x 变成了 1，再执行 g()又把 x 改回了 0。

4.2.6 函数的返回值

函数作为一种模块构件，它与其他模块如何协作、交换信息？我们已经知道，通过函数 调用时的参数传递，可以实现从函数外部向函数内部输入数据。本节讨论函数向外部输出信 息的问题。

在数学中，函数是从定义域到值域的映射，亦即从自变量计算出函数值。编程语言中的 函数原本就是数学函数的模仿物，自然也可以计算出一个结果输出给函数调用者，我们称函 数输出的计算结果为函数的返回值（returned value）。

在前面几章中，我们已多次使用过具有返回值的内建函数和库函数。例如，内建函数 len() 能够接收一个字符串，然后返回该字符串的长度；数学库中的函数 math.sqrt()接收一个 数值，并返回该数值的平方根。我们还看到，带有返回值的函数基本上可以当作一个值来看 待，可以和其他数据一起进行运算，构成表达式。例如：

```
(-b + math.sqrt(b*b – 4*a*c)) / 2*a
range(len("hello"))
x = input("Enter a number:")
```

如何自定义带有返回值的函数呢？Python 语言提供了一条 return 语句用于从函数返 回值，用法如下：

```
def f():
...
return <表达式 1>, ..., <表达式 n>
...
```

其语义是：当 Python 在执行函数 f()时，一旦遇到 return 语句，就终止执行函数，并将 控制返回到函数调用点，同时将各表达式的计算结果返回给调用者。

与 Python 内建函数、库函数一样，带返回值的用户自定义函数可以像一个普通的数据 值一样使用，例如用在表达式中参加运算（当然要求数据类型合法）或者作为赋值语句的右 端为变量赋值。

例如，下面的函数实现了数学函数 f(x) = x2 的功能：

```
>>> def sq(x):
        return x * x
>>> sq(2)
4
>>> print sq(3) + 1
10
>>> a = 4
>>> b = sq(a)
>>> print b
16
```

再看一个例子，下面的 dist()函数能够计算平面上两点间的距离。我们将平面上的点 表示为由横坐标和纵坐标组成的元组(x,y)。根据数学中的距离公式，并利用上面的 sq() 函数，可以写出如下代码：

```
>>> import math
>>> def dist(u,v):
        d = math.sqrt(sq(v[0]-u[0])+sq(v[1]-u[1])) return d
>>> dist((0,0),(4,0))
4.0
>>> dist((0,0),(0,5))
5.0
>>> dist((0,0),(1,1))
1.4142135623730951
>>> dist((1,2),(3,4))
2.8284271247461903
```

如果函数返回值有多个，那么调用者需要使用多个变量来接收函数的返回值。例如下面 的函数 headtail()对一个列表取出头尾元素：

```
>>> def headtail(list):
    return list[0], list[len(list)-1]
>>> headtail([1,2,3,4,5])
(1, 5)
```

调用 headtail 这种返回多个值的函数时，调用者可以利用多变量同时赋值语句来接

收多个返回值，也可以只用一个变量来接收返回值，因为函数返回的“多个值”实际上构成

一个元组。

```
>>> h,t = headtail([1,2,3,4,5])
>>> print h,t
1 5
>>> v = headtail([1,2,3,4,5])
>>> v
(1, 5)
```

函数中的 return 语句通常都出现在函数的末尾，因为函数一般都是执行完所有步骤 之后才能得出计算结果并返回。然而，有时我们希望在函数到达末尾之前就终止执行并返回， 例如当函数检测到不正确的数据时就没有必要继续执行，因为计算下去只能带来错误结果。 下面这个例子检查用户输入（要求是正数），如果不满足要求则退出函数，否则对用户数据 进行处理。代码如下：

```
>>> def f(x):
    if x <= 0:
        print "Positive numbers only, please."
        return
    y = x ** 3
    return y
>>> f(0)
Positive numbers only, please.
>>> f(2)
8
```

最后要说明一点，在 Python 中，任何函数无论是否包含 return 语句，总是要返回一个值的。如果包含 return 语句，自然就返回程序员指定的值；如果不含 return 语句， 则函数总是返回一个称为 None 的特殊对象。如果编程时忘记在函数中用 return 语句返回 值，而调用处又企图使用返回值，则可能出错。例如，假设上面定义的 dist()函数忘了最 后的 return 语句，我们看会带来什么后果：

```
>>> import math
>>> def dist(u,v):
        d = math.sqrt(sq(v[0]-u[0])+sq(v[1]-u[1]))
>>> print dist((0,0),(2,2))
None
>>> print 2 + dist((0,0),(2,2))
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module> print 2 + dist((0,0),(2,2))
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

可见调用 dist()后得到的结果是 None；如果将这个 None 用于表达式中（例中是与 2 相 加）则可能出错，因为对 None 对象并没有定义加法运算。对初学 Python 编程的人来说， 这是容易犯错的地方，所以一定要注意返回值。

4.3 自顶向下设计

采用传统过程式语言进行模块化编程时，主要通过自顶向下方法来进行系统设计。 自顶向下设计也称为逐步求精（stepwise refinement），是将一个系统逐层分解为子系统的设计过程。首先，对整个系统进行概要设计，指明构成系统的顶层子系统有哪些，注意在 此并不给出各个子系统的细节。其次，对每个子系统重复这个设计过程，即再将每个子系统 分解为下一层的子系统。就这样不断细化每个子系统，直至子系统的功能足够简明，可以直 接编码实现为止。

自顶向下设计具有两个特征：第一，要求设计者一开始就对整个系统有清楚的理解，否 则第一步的分解就无法进行；第二，任何子系统在足够细化之前无法开始编码实现，因而必 须等到所有子系统都足够细化，才可能对系统编码实现及测试。

更具体地说，用自顶向下方法编程序时，总是先写主程序，它是由根据系统功能划分而 成的功能子程序组成的。然后再分析每个子程序的需求，如果有必要就继续像主程序一样分 解下去。当划分出来的子程序最终具有非常简单的功能时，就直接编码实现。当所有子程序 都编码实现，整个程序也就实现了。可以相信，由于分解过程总是导致越来越小的程序部件， 最终必然达到“足够简单”的层次，因此不可能无限分解下去。

下面通过一个案例程序来演示自顶向下设计方法。 我们要解决的问题是打印公元某年的年历。要说明一点，为了避免涉及公元历法的一些历史变迁问题，我们对需求做了简化，只要求程序适用于公元 1900 年以后各年份①。相应程 序的规格说明如下：

> ① 程序算法实际上是一般的。将基准日期换成格里高利历的开始日（1582 年 10 月 15 日，星期五）后，很 容易扩展本程序的适用年份范围。

```
程序：calendar
输入：公元年份 year（1900 以后）
输出：year 年年历
输入与输出的关联：根据 year 可以算出相对于 1900 年 1 月 1 日（星期一）总共过去了多少天，按 7 天循环即可
```

4.3.1 顶层设计

根据 calendar 程序的规格说明，很容易设计一个简单的 IPO 模式的算法：首先从用户处 获得年份输入 year，然后计算该年份 1 月 1 日是星期几，最后按特定格式输出年历。我们用 伪代码来表示该算法，如下：

```
输入 year
计算 year 年 1 月 1 日是星期几 输出年历
```

这个算法属于高层设计，其中第二、第三两个步骤都不是一目了然能直接编码实现的， 但我们不妨假设每个步骤都由一个函数实现，从而可以利用这些函数实现程序。

首先，尽管第一个步骤“输入 year”看上去很容易用 input 语句实现，但我们仍然先 用一个顶层模块——函数 getYear()来表示该步骤的实现。函数 getYear()负 责从用户 处获得输入并返回给主程序使用，因此我们将函数的返回值赋值给主程序变量 year。至此， 我们的 calendar 程序取得了第一个进展：

```
def main():
    year = getYear()
```

其次，计算 year 年 1 月 1 日是星期几，这个步骤不是那么显然，但我们仍然假设函数 firstDay()能够实现该步骤，这个函数以 year 作为输入，然后返回一个代表星期几的值（例如，用 0 表示星期天，用 1 到 6 分别表示星期一到星期六）。在主程序中添加一行调用 firstDay()的语句，并将函数返回值赋值给主程序变量 w，这时程序就进展到如下形式： def main():

```
year = getYear() w = firstDay(year)
```

最后一步是输出年历，仍然假设函数 printCalendar()能够实现该步骤，此函数需要用到的信息包括 year 和 w，无需提供返回值。在 main 中添加相应的函数调用语句之后，得到 calendar 程序的完整结构如下：

```
def main():
    year = getYear()
    w = firstDay(year)
    printCalendar(year,w)
```

至此，我们做出了 calendar 程序的顶层设计，将原始问题分解成了三个模块，当然各模块的细节尚不清楚。主程序虽然只有寥寥 3 行，看上去不过是上面的算法伪代码略加细化的 结果，但它确实满足程序规格说明的要求。此外，我们还为对应每个模块的函数声明了函数

名、参数和返回值，这些信息构成了函数的接口（interface）。在 main 这个层次，并不需要关心 getYear()等函数的实现细节，只需要关注它们对于给定的参数能返回预定的数据。 亦即，只关心每个子程序“做什么”，而非“怎么做”。函数接口正是表达“做什么”信息的。

自顶向下设计中经常使用一种设计工具——结构图（或称模块层次图），其中用矩形表 示程序模块，用两个矩形之间的连线表示模块间的调用关系，在连线旁边用箭头和标注来指 明模块之间的界面信息。各模块分别处于不同层次，高层模块是调用模块（或控制模块）， 低层模块是被调用模块（或受控模块）。结构图最顶层就主程序（总控模块）。例如，calendar 程序的顶层设计可以用如图 4.7 所示的结构图来表示。

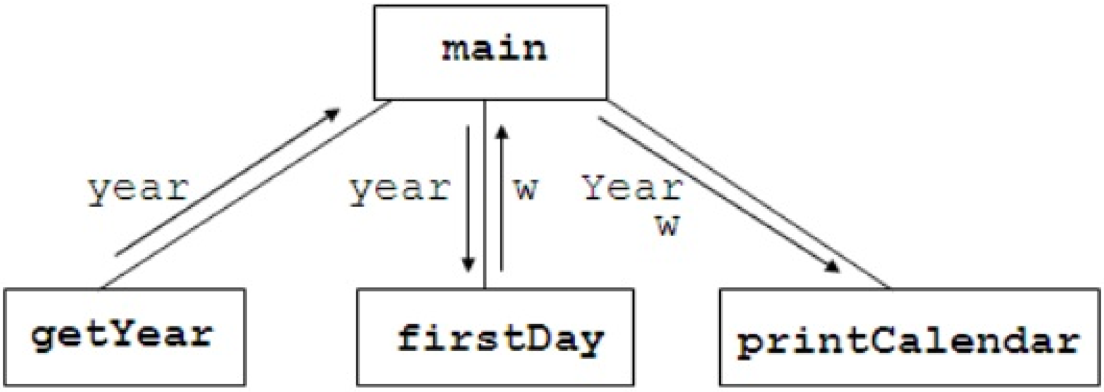


图 4.7 calendar 程序的顶层结构图

在结构图中，越处于下层的模块，其细节程度就越高，即更加精化。

4.3.2 第二层设计

接下来需要对第二层上的每个模块进行精化。

首先看 getYear 函数。这个函数的功能只是输入年份数据，可以直接用 Python 的基本 语句实现，无需分解为新的功能模块。具体代码如下：

```
def getYear():
    print "This program prints the calendar of a given year."
    year = input("Please enter the year (after 1900): ")
    return year
```

接着考虑 firstDay 函数的设计。这个函数的功能是计算 year 年 1 月 1 日是星期几，因为年历是按星期来组织每一天的显示位置的，而只要知道 1 月 1 日的显示位置，其后所有日期的显示位置也就确定了。

在 calendar 程序的规格说明中说明了，我们以 1900 年 1 月 1 日（星期一）作为基准日， 只要算出 year 年 1 月 1 日距离基准日的天数，就能知道这一天是星期几。因为从基准日开始，过 1 天是星期二，过 2 天是星期三，…，过 6 天是星期日，过 7 天又是星期一，…。一般地，过 n 天是星期(n+1)%7（值为 0 表示星期天）。

那么，从基准日到 year 年 1 月 1 日总共过了多少天呢？只需一点常识，就能得出下面 的公式：

```
(year – 1900) * 365 + k
```

其中 k 是从 1900 到 year（不含）之间的闰年个数。

看上去闰年个数 k 还不清楚如何求得，我们按惯例假设一个新函数 leapyears()能够

返回所需的 k。于是可以设计 firstDay 函数如下：

```
def firstDay(year):
    k = leapyears(year)
    n = (year – 1900) * 365 + k
    return (n + 1) % 7
```

最后考虑 printCalendar 函数的设计，该函数的任务是在合适的位置按日历格式显示一年 12 个月的日历。由于问题有点复杂，我们照例进行任务分解。12 个月的日历输出显然可以用一个 for 循环来实现，循环体是显示一个月日历的代码。每个月需要先打印标题（月份和星期的名称），然后再打印日期，假设函数 heading()和 oneMonth()分别执行这两个任务，则 printCalendar 的代码如下：

```
def printCalendar(year,w):
    print
    print "=========== " + str(year) + " =========="
    first = w
    for month in range(12):
        heading(month)
        first = oneMonth(year,month,first)
```

函数体的第一行用于打印年份信息，接下去是打印 12 个月的日历的 for 语句。打印每个月 的日历需要知道该月 1 日是星期几。printCalendar 的参数 w 是前面算出来的 1 月 1 日 的星期信息，2 月到 12 月的 1 日则由 oneMonth 函数返回至此，我们完成了第二层设计， 可以用图 4.8 中的结构图表示到目前为止的设计结果。注意，为简明起见，图中省略了各模 块之间的界面数据。

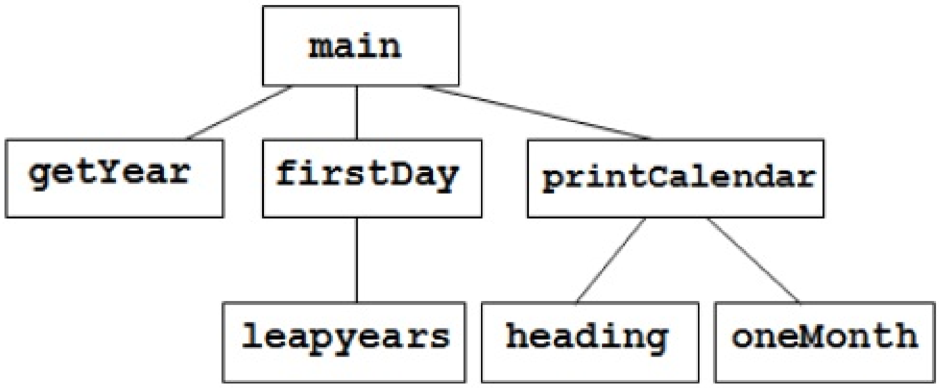


图 4.8 calendar 程序的第二层结构图

4.3.3 第三层设计

首先考虑函数 leapyears 的实现，该函数的功能是计算从 1900 到 year（不含）之间 的闰年个数。这可以用逐年检验的方法来实现①：对从 1900 到 year-1 的每一年，测试该 年是否闰年，如果是则为计数变量 count 加 1。于是得到如下代码：

```
def leapyears(year): count = 0
for y in range(1900,year):
    if y%4 == 0 and (y%100 != 0 or y%400 == 0):
        count = count + 1
    return count
```

其中 if 语句的布尔表达式是根据闰年的规定得到的：年份能被 4 整除并且不能被 100 整除（除非该年能被 400 整除）。

再考虑函数 heading 的实现，该函数用于打印每个月日历的标题部分（月份和星期名 称）。我们将月份名称放在一个列表中，然后通过传递给 heading 函数的月份值作为索引 来查找月份名称。代码如下：

```
def heading(m):
    months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sept","Oct","Nov","Dec"]
    print " %s " % (months[m])
    print "Mon Tue Wed Thu Fri Sat Sun"
```

> ① 如果从公元 1 年算起，到 year 年为止的闰年个数可用公式 year/4 ? year/100 + year/400 计算。

第三层的最后一个函数是 oneMonth()，其功能是输出一个月的日历。由于日历输出要求在合适的位置上显示合适的日期，这个用于输出的子程序反而是整个程序最费功夫的部 分。为了安排日历布局，需要了解每月 1 日是星期几和每月有多少天，还需要确定何时换行显示。我们采用一个长度为 6×7=42 的列表①作为日历布局框架（每行 7 天，一个月最多占用 6 行），只需将一个月的每一天存入这个框架的合适位置，然后输出这个列表即可。图4.9 是日历框架的示意图。

Mon	Tue	Wed	Thu	Fri	Sat	Sun

图 4.9 每个月的日历布局 由于问题有点复杂，我们再次分解任务，用三个子程序来实现 oneMonth()：days() 函数计算该月份的天数，layout()函数用于布置该月每一天在日历框架中的位置， printMonth()用于输出日历。即：

```
def oneMonth(year,month,first):
    d = days(year,month)
    frame = layout(first,d)
    printMonth(frame)
    return (first + d) % 7
```

oneMonth 函数有三个参数：year 表示年份，month 表示月份，first 表示该月 1 日是星期几（0～6）。对于一月份，first 由上层模块 printCalendar 的参数 w 提供； 对于其他月份，first 可由上一个月的 first 和天数确定，因此我们让 oneMonth 在打印 本月日历后返回下个月 1 日的星期序号。

> ① 使用二维列表或许会更直观。

设计至此，结构图演变为图 4.10 所示的情况。

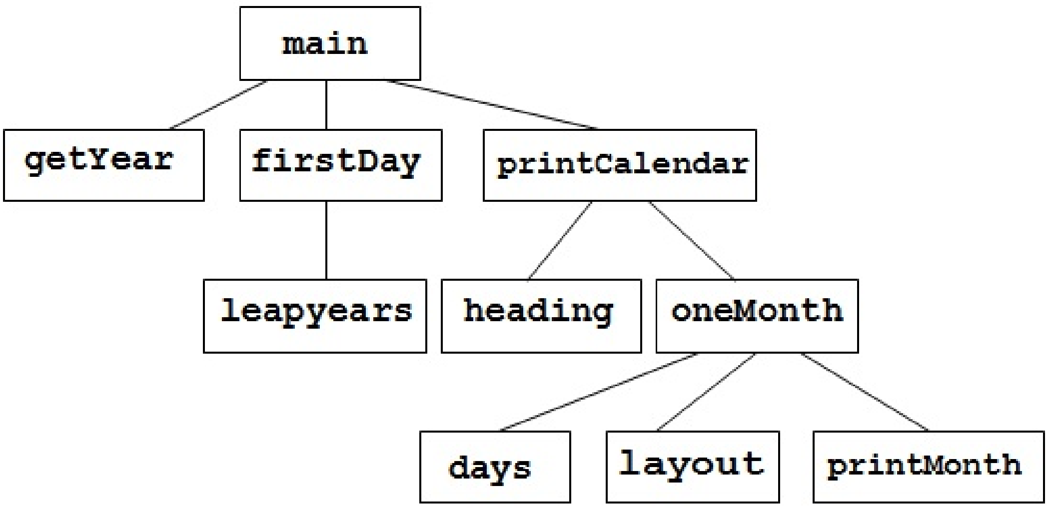


图 4.10 calendar 程序的第三层结构图

4.3.4 第四层设计

先考虑 days 函数的实现。我们将每个月的天数放在列表中，然后通过月份进行索引即 可得到该月天数。要注意有个特殊情形，即闰年 2 月份。这时应当为天数多加 1 天。代码如 下：

```
def days(y,m):
    month_days = [31,28,31,30,31,30,31,31,30,31,30,31]
    d = month_days[m]
    if (m == 1) and (y%4 == 0 and (y%100 != 0 or y%400 == 0)):
        d = d + 1
    return d
```

接着考虑函数 layout 的实现。本函数根据 first 和 d，将每一个日期填入日历框架（图 4.9）。

```
def layout(first,d):
    frame = 42 * [""]
    if first == 0:
        first = 7
    j = first - 1
    for i in range(1,d+1):
        frame[j] = i
        j = j + 1
    return frame
```

最后实现 printMonth 函数。日历布局已经保存在列表 frame 之中，函数要做的事情就是将列表成员打印出来。其中的关键是掌握好换行的时机，采用了日历框架后这一点变得很简单，只需每输出 frame 的七个成员就换行一次。代码如下：

```
def printMonth(frame):
    for i in range(42):
        print "%3s" % (frame[i]),
        if (i+1)%7 == 0:
            print
```

至此，我们为 calendar 程序设计的所有模块都已实现。

4.3.5 自底向上实现与单元测试

自顶向下设计设计是创建层次化的模块结构的过程，而从实现的角度看，我们又是采取了相反的过程，即自底向上的实现。从结构图的底层开始实现每一个函数，然后上一层模块 自然得到实现。就这样自底向上，直至主程序得到完全的实现。

在模块化编程中，测试程序最适合采用单元测试技术，即先分别测试每一个小模块，然 后再逐步测试较大的模块，直至最后测试完整程序。以 calendar 程序为例，当我们实现了 days(y,m)函数后，就应该来测试此函数是否能完成预定的功能——返回 y 年 m+1 月有多 少天。我们可以将 days(y,m)的定义存入一个模块文件（假设文件名是 moduletest.py）， 然后导入该文件并测试函数。下面是测试 days 函数的一个会话过程：

```
>>> from moduletest import days
>>> days(1900,0)
31
>>> days(1900,1)
28
>>> days(1900,11)
31
>>> days(2000,1)
29
>>> days(2012,1)
29
>>> days(2012,10)
30
```

注意，测试时应当使测试数据尽量覆盖所有关键情形。在我们的测试例子中，测试了合 法数据的边界情形 1900 年 1 月，也测试了 1900 年 2 月（这个年份虽然能被 4 整除但却不是闰年），还测试了 2000 年（能被 400 整除）是否闰年。所有测试结果都表明这个函数实现正确。

单元测试技术独立地测试每一个函数，这样能更容易定位程序错误。如果较小模块都正 确，那么由它们组成的较大模块出现错误的可能性也就较小。最终测试完整程序时，就更有 希望通过测试。

最后，为了完整起见，我们将前面所有的代码汇集起来列在下面。

【程序 4.8】calendar.py

```
# calendar.py
def getYear():
    print "This program prints the calendar of a given year."
    year = input("Please enter a year (after 1900): ")
    return year
def firstDay(year):
    k = leapyears(year)
    n = (year - 1900) * 365 + k return (n + 1) % 7
def leapyears(year): count = 0
    for y in range(1900,year):
        if y%4 == 0 and (y%100 != 0 or y%400 == 0):
            count = count + 1
    return count
def printCalendar(year,w): print
    print "=========== " + str(year) + " =========="
    first = w
    for month in range(12):
        heading(month)
        first = oneMonth(year,month,first)
def heading(m):
    months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sept","Oct","Nov","Dec"]
    print " %s " % (months[m])
    print "Mon Tue Wed Thu Fri Sat Sun"
def oneMonth(year,month,first): d = days(year,month)
    frame = layout(first,d)
    printMonth(frame) return (first + d) % 7
def days(y,m):
    month_days = [31,28,31,30,31,30,31,31,30,31,30,31]
    d = month_days[m]
    if (m == 1) and (y%4 == 0 and (y%100 != 0 or y%400 == 0)):
        d = d + 1
    return d
def layout(first,d): frame = 42 * [""]
    if first == 0:
        first = 7
    j = first - 1
    for i in range(1,d+1):
        frame[j] = i
        j = j + 1
    return frame
def printMonth(frame):
    for i in range(42):
        print "%3s" % (frame[i]),
        if (i+1)%7 == 0:
            print
def main():
    year = getYear()
    w = firstDay(year)
    printCalendar(year,w)
main()
```

图 4.11 显示了本程序的一次运行结果，可见程序是正确的（注意 2012 是闰年）。当然，输出的日历在格式上还可以美化，例如将两三个月的日历放在同一排上之类。读者不妨自行设计修改。

```
C:\Python27\python.exe
This program prints the calendar of a given year.
Please enter a year (after 1900): 2012

=========== 2012 ==========
            Jan
Mon Tue Wed Thu Fri Sat Sun
                          1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
            Feb
Mon Tue Wed Thu Fri Sat Sun
          1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29

            Mar
Mon Tue Wed Thu Fri Sat Sun
              1   2   3   4
  5   6   7   8   9  10  11
 12  13  14  15  16  17  18
```

图 4.11 calendar 程序的运行示例

4.3.5 开发过程小结

calendar 程序的完整开发过程，展示了自顶向下设计方法的强大能力。当面临一个复杂 问题而感到无从下手的时候，可以尝试将原始问题分解为若干个子问题，然后再去考虑每个 子问题的解决方案。这个分解过程可以重复进行，从结构图的顶层开始，自顶向下逐步求精， 直至得到所有子问题的精确代码。

自顶向下设计过程可以概括为以下四个步骤：

（1）将问题分解为若干子问题；

（2）为每个子问题设计一个函数接口；

（3）将原问题的算法用各子问题对应的函数接口来表达；

（4）对每个子问题重复（1）～（3）的过程。 经过以上步骤，高层的抽象接口在低层逐步得到细化，最终到达可以直接用 Python 基

本语句实现的层次。 自顶向下设计是编写复杂程序的重要工具，虽然这种方法会导致很多小模块（函数），看上去设计起来有点麻烦，但这其实是事半而功倍的方法。事实上不采用模块化方法是不可 能设计出复杂系统的。

模块化设计和单元测试都是分离关注点原则的具体体现，前者使我们能够设计复杂程 序，后者使我们能够调试复杂程序。作为初学者，应当不断地实践模块化方法，让模块化思 想和方法变成自己的本能思维方式。

最后要说明一点，自顶向下设计是非常强大的编程技术，但并非唯一的编程技术，有时 这种设计方法并不可行。例如，自顶向下设计的第一步是对整个系统进行任务分解，然而在 开发某些应用时，可能无法对整个系统的需求先有充分的理解，只能随着开发的进行，逐渐 获得对系统的理解，这时就不可能采用自顶向下设计。

本书后面还会介绍其他程序设计方法，比如原型方法、面向对象设计等等。程序设计是一个创造性的过程，并不存在什么唯一正确的方法或者一成不变的规则。好的开发者应当掌 握多种设计方法。虽然通过读书学习可以了解程序设计技术，但更重要的是通过实践来掌握 在什么场合应用以及如何应用这些方法。

4.4 Python 模块*

模块这个术语通常用于泛指相对独立的程序单元，Python 语言中的模块既有这种一般 含义，还有其特定的含义。

4.4.1 模块的创建和使用

在 Python 语言中，模块对应于 Python 程序文件，即每个 Python 程序文件就是一个模块。模块是 Python 程序的最高层结构单元，用于组织程序的代码和数据，以便能被同一程

序的其他模块甚至被其他程序重用。一个模块可以导入其他模块，导入后就可以使用其他模块中定义的函数、类等对象。

用模块作为程序的结构单元，至少有三个作用：

（1）代码重用：将代码保存在能持久存在的文件中，就不会像在 Python 交互环境中键 入的代码那样随着退出 Python 而消失。模块中的代码可以多次加载运行，也可以被多个程 序使用。

（2）名字空间：模块是 Python 的最高层程序结构单元，在模块中定义的所有名字（函 数名、 类名等）是局部于本模块的，与模块外部不会发生同名冲突。要想使用一个模块定义 的名字，唯一途径就是导入该模块。

（3）实现共享：模块对于实现全系统范围内代码和数据的共享也是很有用的，被共享 的东西只需保存一个副本。例如，如果需要为多个函数或模块提供一个全局对象，则可以将 它的定义置于一个模块中，然后其他使用者可以导入该模块，从而共享使用全局对象。

Python 模块很容易创建。只要使用任意的文本编辑器，键入一些 Python 语句并保存 为.py 文件，就得到一个 Python 模块。

为了使用 Python 模块中定义的对象，必须用 import 或 from 语句导入模块。import 的功能是导入模块整体，导入后为了访问模块定义的对象，必须在对象前加上模块名作为前 缀。例如，假设模块 mymod 中定义了我们需要用到的函数 func()，那么可以这样导入：

```
import mymod mymod.func()
```

另一种导入语句是 from 语句，用于导入模块中定义的特定名字（用*可以导入所有名字）。使用时不需要加上模块名作为限制。例如：

```
from mymod import func func()
```

注意，导入模块后，模块名就能像普通 Python 变量一样在程序中使用。因此模块名必须符合 Python 命名规则。

4.4.2 Python 程序架构

简单程序可以只用一个程序文件实现，但对绝大多数 Python 程序，一般都是由多个源 文件（即模块）组成的，其中每个源文件都是包含 Python 语句的文本文件。

具体来说，Python 程序通常是由一个顶层主文件和多个模块文件组成的。顶层主文件 定义了程序的主控制流，是执行应用程序时的启动文件；模块文件则是“工具”库，用于汇 集顶层文件和其他模块需要用到的函数等部件。顶层文件使用模块文件中定义的工具来完成 应用功能，同时一个模块也可使用别的模块定义的工具。

模块文件一般不能直接执行，模块中只是定义了很多工具给其他模块使用。Python 中 通过导入模块来使用该模块定义的工具。图 4.12 描绘了一个由三个文件（a.py、b.py 和 c.py）组成的 Python 程序，其中 a.py 是顶层文件，b.py 和 c.py 是模块。b.py 和 c.py 一般不能直接执行，该程序的执行只能通过 a.py 来启动。

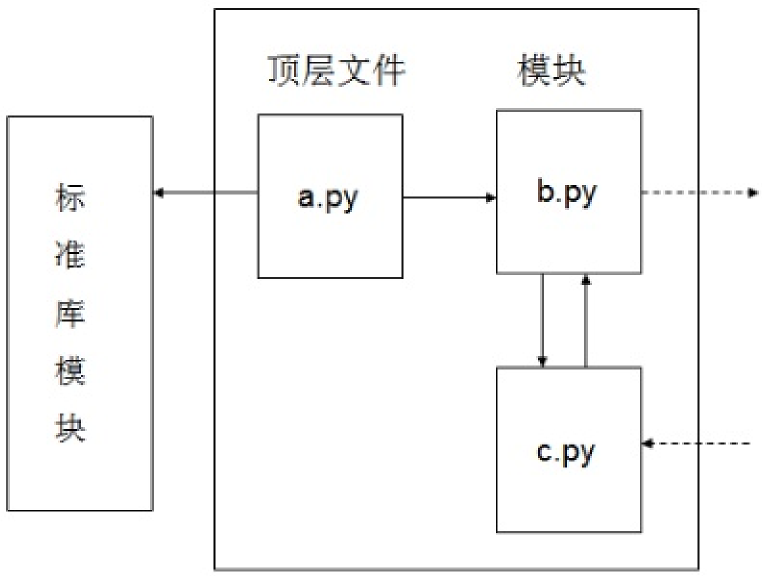


图 4.12 Python 程序架构

假设文件 b.py 中定义了一个函数 hello 给外部使用：

```
def hello(person):
    print "Hello", person
```

再假设 a.py 正好需要使用 hello()，为此可以在 a.py 中导入模块 b，然后调用 hello()：

```
import b
b.hello("Lucy"')
```

其中的导入语句使得 a.py 能够访问 b.py 中顶层代码所定义的所有名字（这里只有 hello）。a.py 的第二条语句调用模块 b 中定义的函数 hello，其中 b.hello()这种“点 表示法”其实是面向对象的表示法，b 是一个模块对象，hello 则相当于 b 对象的一个属 性。b.hello 就等于说“对象 b 中的属性 hello 的值”，这个值恰好是一个可调用的函数， 因此可以传递一个字符串参数"Lucy"'给它。

任何模块文件都可以从任何其他模块文件导入定义，例如文件 a.py 可导入 b.py， b.py 也可以导入 c.py。导入链条可以任意深入下去：a 导入 b，b 导入 c，c 导入 b，等 等。

除了作为最高层结构单元，模块还是代码重用的最高层形式。例如，如果很多模块都需 要使用函数 b.hello，那我们可以在别处导入 b.py，从而达到代码重用的目的。

4.4.3 标准库模块

应用程序要导入的模块大多来自 Python 语言提供的标准库。Python 标准库实现了很多 常见功能（如操作系统功能、GUI 构建、网络与互联网编程等），对应用程序设计提供了强 大的支持。标准库并不是 Python 语言本身的一部分，而是由专业程序员预先编好并随语言 提供给用户使用的。Python 的标准安装都会自动安装标准库。

如果想了解随着 Python 安装的标准库中有哪些模块，可以使用 Python 的联机帮助命令。 在 Python 解释器提示符下键入 help()，可以进入联机帮助环境：

```
>>> help()
Welcome to Python 2.7! This is the online help utility.
......
help>
```

省略号是 Python 打印的一些说明信息。help>是帮助系统的提示符，可以在这个提示符下 输入想了解的主题，Python 就会给出有关主题的信息。例如输入 modules 可以得到安装的 所有模块的信息：

```
help> modules
Please wait a moment while I gather a list of all available modules... AppClass1 asynchat
......
help>
```

输入某个模块的名字可以获得该模块的信息，例如：

```
help> math
Help on built-in module math:
NAME
math
FILE
(built-in)
DESCRIPTION
This module is always available. It provides access to the mathematical functions defined
FUNCTIONS
acos(...)
acos(x)
Return the arc cosine (measured in radians) of x.
```

从系统显示的信息中我们了解到 math 模块中 acos 函数的意义和用法。

在 Python 中，要想编写有用的或有趣的应用程序，往往并不需要自己写很多代码，标

准库中有大量的现成代码可用。读者需要时可自行查阅有关 Python 标准模块的资料，以求事半功倍。

4.4.4 模块的有条件执行

有些 Python 模块是可以直接执行的，一般称为程序或脚本；而另一些 Python 模块中只 包含一些函数定义，本身并没有主程序入口，因而不能执行。标准库就属于后一种模块。有 时我们希望创建一种混合式的模块——既可以作为独立执行的程序，又可以作为被其他程序

导入的库。在 Python 中，混合式模块可以通过在程序入口前加上特定条件而实现。 如所熟知，我们一般都在程序文件的最后加上启动程序的一行语句：

```
main()
```

这是对程序入口（主函数 main）的调用，没有这一行，程序文件就不是可执行的文件。这 就是直接执行的模块文件，在窗口系统中用鼠标双击即可启动程序。

Python 在导入一个模块的时候会执行模块中的每一行语句，执行函数定义语句 def 时 就创建相应的函数但并不执行，而最后遇到启动程序的 main 时就启动了整个程序。有时我 们希望导入模块时不要执行整个程序，例如交互环境下测试程序时，通常的做法是先导入模 块，需要执行代码时才去调用 main 或其他函数。要想只导入不执行，当然可以删掉程序入 口 main()，但这又会失去双击执行程序的可能。两全其美的做法是在主程序入口 main 之 前加个条件：

```
if <条件>:
    main()
```

意思是当条件满足时启动程序，否则不启动程序。问题是条件怎么写？

如果是用 import 导入模块，Python 会将该模块的一个特殊变量 name 的值设置为 模块的名字。例如：

```
>>> import math
>>> print math.__name__
math
```

第一行导入模块 math，并将 math 的变量 **name** 设置为'math'。第二行显示了这个变量的值。

但如果是直接执行模块（如双击模块文件图标等），Python 则将模块的特殊变 量 name 设置为字符串'main'。因此可以通过特殊变量 name 的值来判断模块是被 导入的还是被直接执行的。根据这个底层细节，我们可以将程序文件的最后一行改成：

```
if __name__ == '__main__':
    main()
```

这样就能确保当程序是直接执行时，main 能启动；当程序是被导入时，忽略 main。

4.5 练习

1. 什么是模块化设计？

2. 模块有哪些特点？

3. 什么是分离关注点原则？

4. 子程序的创建和调用涉及哪些内容？

5. 程序中为什么引入函数？

6. 什么是形式参数和实际参数？参数传递的过程是怎样的？

7. 什么是变量的作用域？什么是全局变量与局部变量？

8. 函数的参数与局部变量的异同是什么？

9. 函数调用时的控制流是如何转移的？

10. 什么是自顶向下设计？主要分为哪几个步骤？

11. 为具有下列主函数的程序画出结构图的顶层。

```
def main():
    printIntro()
    length, width = getDimensions()
    amtNeeded = computeAmount(length,width)
    printReport(length, width, amtNeeded)
```

12. 请写出五个 Python 标准库中的模块名称及其主要功能。

13. 考虑函数：

```
def cube(x):
    answer = x * x * x
    return answer
```

（1）这个函数的功能是什么？

（2）设 y 是一个变量，如何用 cube 函数去计算 y3？

（3）考虑下面这个程序片段：

```
answer = 4
result = cube(3)
print answer, result
```

由于 cube 将 answer 赋值成了 27，所以输出应该是 27 27，对不对？为什么？

14. 设计程序：在屏幕上打印歌曲《歌唱祖国》的歌词①。

15. 设计程序：给定两个平面上的点 p1 和 p2（用元组表示），函数 slope(p1, p2)返回 通过 p1 和 p2 的直线的斜率，函数 intercept(p1, p2)返回该直线在 y 轴上的截距。

16 改写本章中的 calendar 程序，使输出更美观（例如让每三个月的日历输出在同一排上）。

17. 采用自顶向下设计方法编写程序：在屏幕上打印三角函数 y = sin(x)的图像。

18. 重做第 3 章的程序设计练习题，尽量使用函数来封装计算。

> ① 歌词参见 http://baike.baidu.com/view/252108.htm

第 5 章 图形编程

在现实中，人们经常利用直观的图形来表达抽象的思想，图形可以帮助人们设计产品、 理解数据、洞察规律。同样地，在用计算机解决问题时，也经常需要绘制图形。有些应用本 身就是图形图像应用，而另一些应用只是利用图形来使计算可视化。本章主要介绍 Python 图形编程。由于图形是复杂数据，对复杂数据的表示和操作最适合采用面向对象方法，因此 本章还将初步介绍面向对象的基本概念①。

5.1 概述

实际应用中经常需要利用图形、图像和动画。例如，在大量数据的统计与分析中，仅仅 算出数学期望、标准差之类的统计指标，并不能使普通人很好地理解数据；但是如果画出直 方图、趋势曲线之类的图形，就能使人们洞悉数据所蕴涵的意义。又如，假设小学教师希望 向学生讲授太阳、地球和月亮三者之间的位置和运动的知识，如果他完全用文字语言来表述 天文知识，恐怕小学生会很难理解；但如果他用图形或动画来演示，相信小学生立刻就能明 白三个天体间的复杂运动。

可见，计算机图形能够帮助我们更好地解决问题，是非常重要的编程技术。本节对图形 编程的意义进行简单讨论，从下一节开始具体介绍 Python 的图形编程。

5.1.1 计算可视化

随着计算机硬件和软件技术的发展，计算机图形技术越来越成熟，如今已经在各行各业中得到了广泛应用。有一些应用本身的任务就是绘制图形，例如制作动画片、艺术设计之类；还有一些应用不以绘图为目的，但会利用图形来辅助完成任务，例如统计应用的目的是计算 各种数值指标，但常用图形来直观地展示统计结果。

可视化（visualization）是指将抽象事物和过程转变成视觉可见的、形象直观的图形图 像表示。计算可视化就是在用计算机解决问题的过程中，使用图形图像来表达数据和操作。 图形图像所具有的直观性能使我们更有效地传达信息，即使这信息是非常抽象的。在历史上， 用可视的图形图像来展现信息是很常见的，在有文字之前人类就用图画表达信息，甚至文字 本身也是从图形发展而来的。如今，计算机图形技术为计算可视化提供了强大的支持，促进 了可视化计算在科学、工程、教育等领域的广泛应用。应用中常见的图形包括柱状图、直方图、散点图、网络图、流程图、树、地图、图像、动画等等。

科学可视化

可视化术语最初是指科学可视化，也就是将科学与工程计算、实验中的大规模数据用直 观的计算机图形图像呈现出来，以便人们理解数据、增强对事物现象的认识和对内在规律的 洞察。

计算机图形技术从诞生起就被用于研究科学问题，如今科学可视化在物理、化学、医学、 空间科学等领域得到了大量应用。通过科学可视化，我们看清了台风的形成、太空飞行器的 活动、分子原子的结构以及人体内部的病灶，并能将纯抽象的概念和构造在 3 维空间中展现 出来②。

工程设计可视化 在工程领域，可视化被计算机辅助设计和制造（CAD/CAM）系统广泛地使用。无论是土木工程还是机械工程、电子工程，设计人员借助计算机图形软件和设备从事产品设计工作， 例如利用计算机自动生成设计图，对设计图进行编辑、缩放、旋转，对不同方案进行比较和 优选等等。此外，可视化还可以使工业过程控制、系统模拟、生产管理等任务以直观的方式 进行，以实现更有效的控制和管理。

> ① 本书第 7 章详细介绍面向对象编程。
>
> ② 美国 Science 杂志和 NSF 每年都举办“国际科学与工程可视化挑战赛”，建议读者搜索获奖作品看看。

数据可视化

数据可视化是指利用计算机图形学和图像处理技术，将海量数据转化为数据图像，以便 帮助人们直观地观察数据。对于多维数据（例如人事数据包含姓名、性别、学位、收入等多 种维度），利用数据图像还可以从不同的维度观察数据。一般认为，数据、信息、知识构成 由低到高的三个层次，因此从数据可视化可以进而发展到更高层次的信息可视化（发现数据 中隐藏的模式、关联或趋势）和知识可视化（促进知识的传播）。

图形用户界面

可视化最常见的应用当属图形用户界面（GUI）。计算机软件的用户界面负责支持用户 与计算机进行交互。早期软件的用户界面都是文字式的，用户在屏幕上看到的输出都是文本 信息，并且只能通过键盘输入文本命令来控制软件执行。如今的软件几乎都具有图形用户界面，屏幕上展现给用户的是各种可视的图形元素，如窗口、图标、按钮和菜单等等；而用户可以使用鼠标来点击图形元素以控制程序的执行。这样的 GUI 软件使用起来非常直观、高效，具有所谓的“用户友好性”。本书第 8 章将详细介绍 GUI 编程。

除了上述领域之外，人们还在教育领域利用可视化创建现实中难以见到的事物（如血液 循环系统、化合物分子、恐龙等）的图形图像，以使教学形象直观；在刑事侦查领域利用可 视化重建犯罪现场、绘制案犯相貌；在娱乐领域利用可视化制作计算机电影特效、动画；等 等。

总之，计算机图形技术极大地增强了人们利用计算机解决问题的能力。因此，学习图形 编程是非常重要的。

5.1.2 图形是复杂数据

图形编程就是编写能创建和处理图形的程序。从一般的意义上说，图形也是数据，只不过与数值、字符串、列表等类型的数据相比，图形数据是非常复杂的数据。 首先，一个图形包含的信息是复杂的。例如，一个圆形需要用一个圆心和一个半径来定义。半径可以用一个简单的数值来表示，但圆心（平面上的一个点）却需要用两个数值型坐 标组成的元组来表示。这还只是大家在平面几何里认识的圆形，在实际的图形应用中还会考 虑圆形内部和轮廓线的颜色、轮廓线线条的粗细等问题，其中颜色又是由红绿蓝三种颜色分 量构成的复杂数据。可见图形确实是很复杂的数据。

其次，对图形的处理是复杂的。对数值，可以加减乘除；对字符串，可以求子串或连接 两个串；对列表可以取成员或求长度。对一个圆形，能做什么呢？数学中会去求面积、求周 长，但在图形应用中更有意思的操作是改变颜色、移动到另一个位置等，这些操作相对于加 减乘除之类显然复杂得多。

那么，在编程语言中如何表示图形这一类的复杂数据、如何操作这类复杂数据呢？编程 语言一般没有内建的图形数据类型和对图形的操作，但会提供标准图形库用于支持图形编 程。本章将介绍如何使用 Python 语言的标准图形库 Tkinter 来进行图形编程。在介绍 Tkinter 之前，需要先简要介绍对象的概念，因为现代的图形库一般都是采用面向对象技术来实现图 形数据类型的。

5.1.3 用对象表示复杂数据

程序是对数据进行操作的过程，因此数据表示和操作过程是编程时要考虑的两大问题。 我们已经熟悉用编程语言提供的数据类型来表示数据，例如用字符串表示雇员姓名，用整数表示年龄，用浮点数表示工资等。对于某些稍微复杂一点的数据我们也有适合的数据类 型来表示，例如雇员名单可以用一个字符串数据构成的列表来表示。当数据表示确定之后， 我们接着用各种数据类型所支持的数据操作来处理数据，例如对于工资数据可以执行加减乘 除操作，对于姓名数据可以分别抽取出姓和名。

先考虑数据的表示，然后再考虑对数据的操作，这就是迄今为止我们在编程序时常用的 思考方式。在这种思考方式下，数据和对数据的操作被看作是两件相互分离的事情，因而可 以分别考虑。例如，在一元二次方程求解程序中，我们先获得所需的数据（方程系数 a、b、c），然后才去考虑对这些数据的操作过程，即先计算判别式的正负，再去求方程根。

然而还有另一种思考方式，那就是将数据和对数据的操作视为不可分离的，并将两者组 合在一起形成一个实体——对象（object）。显然，对象是对传统“数据”概念的发展：传统 数据只是存储一些信息，而对象中不但存储了一些信息，而且还掌握了对这些信息的操作。 在面向对象术语中，对象的数据称为属性，对象的操作称为方法。

以一个简单数据"Lu Chaojun"为例，在传统观点下，可用字符串类型来表示这个数 据：

```
name = "Lu Chaojun"
```

现在，数据 name 仅仅存储了一个姓名，对这个数据能执行什么操作不由 name 决定，而是由程序的其他部分决定。例如，如果希望按西方习惯将姓放在名的后面显示，则程序中可以对 name 进行如下操作：

```
lastname = name[0:2]
firstname = name[3:]
print firstname,lastname
```

而在对象观点下，我们将把 name 和能对 name 执行的操作相结合，形成一种对象（如图 5.1 所示），该对象 不但存储了 信息 "Lu Chaojun"，而且还拥有 对信息的操 作last_first()、first_last()、first()、last()等。这种对象本质上仍然是 name数据，而且是具有数据操作能力的数据。

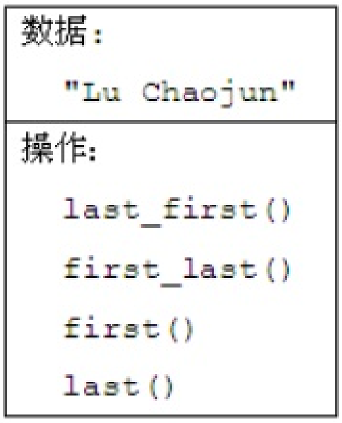


图 5.1 对象：数据与操作相结合

总之，一个对象不但知道一些信息，并且还负责操作这些信息。要想对对象的数据执行特定操作，只需向对象发出请求消息，由对象来执行所需的方法。 对象概念通常并不是用来描述如上例那样的简单数据的。事实上，对象概念主要用于描述复杂数据、设计复杂系统。对象将若干相关数据连同若干操作组合在一起，形成一种结构单元，从而复杂系统可以方便地设计成由许多对象组成，对象之间通过交互、协作来完成系 统功能。

图形应用程序涉及图形这样的复杂数据以及对图形的各种操作，因此非常适合采用面向 对象概念。许多语言的图形库都是面向对象风格的，其中包括我们将介绍的 Python 标准图 形库 Tkinter。

5.2 Tkinter 图形编程

Python 语言自带一个标准模块 Tkinter，这是一个功能强大的图形用户界面工具包，能够用来开发像 Windows 应用程序一样具有窗口、菜单、按钮等图形构件的程序。本章只介绍 Tkinter 中的绘图功能，基于 Tkinter 的 GUI 编程将在第 8 章中介绍。

5.2.1 导入模块及创建根窗口

为了使用 Tkinter 模块中提供的绘图功能，首先要将该模块导入到程序中，就像我们以 前导入 math 模块以使用其中的数学函数、导入 string 模块以使用其中的字符串操作函数一 样。可以用下列两种方式中的任何一种导入 Tkinter：

```
import Tkinter
```

或者

```
from Tkinter import *
```

如我们以前所说，这两种导入方法的差别仅在于以后调用模块中的函数时是否要加上模块名作为前缀。注意，以下我们总是假设使用第二种方式导入 Tkinter 模块。

导入 Tkinter 之后，第二步要做的就是创建一个窗口（称为根窗口），所有图形都是在这 个窗口中绘制的。下列语句创建根窗口并赋值给一个变量 root：

```
root = Tk()
```

接下去就可以在根窗口中绘制图形了。

以下我们将采用交互式环境来演示 Tkinter 的绘图语句，读者可以照样子键入这些语句 并得到和本书图示一样的结果。注意，由于 IDLE 本身是用 Tkinter 写的程序，在 IDLE 中执 行 Tkinter 语句会有问题，因此本章所有交互式演示都是在命令行环境中执行的。另外，演 示中既可以在同一个窗口中连续执行绘图语句，也可以在每次演示新的图形语句时重新创建 根窗口。不管是哪一种做法，为了避免重复，我们总是假设已经执行了下面两条语句：

```
>>> from Tkinter import *
>>> root = Tk()
```

这时可以在屏幕上看到如图 5.2 所示的窗口。



图 5.2 根窗口

根窗口实际上是一个对象，它有自己的属性（如宽度、高度、窗口标题），也有自己的 方法。本章只关注绘图功能，不需要对根窗口进行操作。有关内容可参见第 8 章。

5.2.2 创建画布

为了绘图，首先要有画布。Tkinter 中提供了画布（Canvas），可以在画布上绘制图形、 文本，也可以在上面放置命令按钮等 GUI 构件。画布实际上是一个 Canvas 对象，它包含 一些属性（如画布的高度、宽度、背景色等），也包含一些方法（如在画布上创建图形、删 除或移动图形等）。

创建画布对象的语句模板如下：

```
c = Canvas(<窗口>,<选项 1>=<值 1>,<选项 2>=<值 2>,...)
```

其中 Canvas 是 Tkinter 提供的类（class），所谓“类”其实就和 int、float 等一样是数据类型，只不过不是 Python 语言的内建类型，而是 Tkinter 模块带来的扩展类型。Canvas就像一个制造画布的工厂，每次执行 Canvas()都能制造出一个画布对象。参数<窗口>表示画布所在的窗口，诸<选项>=<值>为画布对象的选项（即属性）进行赋值。总之，整条语句创建一个 Canvas 对象，对该对象的数据进行设置，并将该对象赋值给变量 c（更准确的 说法是变量 c 引用该对象）。

画布的常用选项包括 height（画布高度）、width（画布宽度）和 bg（或 background， 画布背景色）等，需要在创建画布对象时进行设置。创建画布对象时如果不设置这些选项的 值，则各选项取各自的缺省值，例如 bg 的缺省值为浅灰色。画布对象的所有选项都可以在 创建以后的任何时候重新设置。

下面的语句在根窗口 root 中创建一个宽度为 300 像素①、高度为 200 像素、背景为白 色的画布：

```
>>> c = Canvas(root,width=300,height=200,bg='white')
```

注意，虽然至此已经创建了画布对象，但在根窗口中并没有看到这块白色画布，这就好 比从商店买来了画布，但还没有铺到桌子上一样。为了让画布在窗口中显现出来，还需要执 行如下“布置画布”的语句②：

```
>>> c.pack()
```

现在，我们在屏幕上看到原来的根窗口（背景色为浅灰色）中放进了一个 300x200 的白色画布。如图 5.3 所示。



图 5.3 放入画布后的根窗口

这里需要对 c.pack()所用到的“点表示法”加以说明。过去，当我们编写了一个函数f()来操作数据 x，传统的表示法是 f(x)。而在面向对象编程中，数据和操作被结合在一 起形成了对象，如果要对对象中的数据执行操作，通常采用点表示法——“对象.操作”。在 c.pack()中，变量 c 表示一个 Canvas 对象，pack()是 Canvas 对象能够响应的一个方 法，故 c.pack()就表示向对象 c 发出执行 pack()方法的请求。

> ① 像素（pixel）是能显示的最小图像单元，通俗说就是一个点。数字图像是由很多点组成的。
>
> ② 在窗口中布置各种构件需要使用布局管理器，这里的 pack()就是一种布局管理器。详见第 8 章。

坐标系

创建了画布，接下来就可以在画布上绘制各种图形了。为了在绘图时指定图形的绘制位 置，Tkinter 为画布建立了坐标系统。画布坐标系以画布左上角为原点，从原点水平向右为 x 轴，从原点垂直向下为 y 轴（图 5.4）。

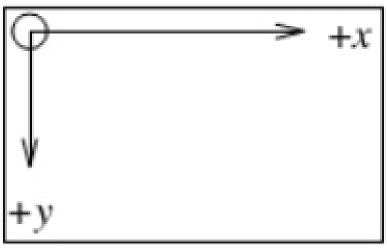


图 5.4 画布的坐标系统

坐标如果以整数给出，则度量单位是像素，例如左上角原点的坐标为(0,0)，300x200的画布的右下角坐标为(299,199)。像素是最基本、最常用的长度单位，但 Tkinter 也支持以字符串形式给出其他度量单位的长度值，例如"5c"（5 厘米）、"50m"（50 毫米）、"2i"（2 英寸）等。

图形项的标识

一个画布上可能创建多个图形项①，因此需要有办法来标识、引用其中某个图形项，以 便对该图形项进行处理。画布上的图形项有两种标识方式：

- 标识号：创建图形项时 Tkinter 自动为图形项赋予一个唯一的整数编号。
- 标签：图形项可以与字符串型的标签（tag）相关联，每个图形项可以与 0、1 乃至 多个标签相关联，而同一个标签可以与多个图形项相关联。

标签相当于为图形项命名，只不过一个图形项可以有多个名字，而且不同图形项可以有 相同的名字。为图形项指定标签的方法有三种：一是在创建图形时利用选项 tags 来指定， 可以为 tags 选项提供单个字符串（单个名字），也可以提供一个字符串元组（多个名字）； 二是在图形创建之后，任何时候都可以利用画布的 itemconfig()方法来设置；三是利用 画布的 addtag_withtag()方法来为图形项增添新标签。假设我们已经创建了画布 c，则 可以执行：

```
>>> r1 = c.create_rectangle(20,20,100,80,tags="#1")
>>> r2 = c.create_rectangle(40,50,200,180,tags=("myRect","#2"))
>>> c.itemconfig(r1,tags=("myRect","rectOne"))
>>> c.addtag_withtag("ourRect","rectOne")
```

> ① 每个图形项可以理解成一个图形对象（有自己的属性和操作），不过 Tkinter 没有采用为每种图形提供单 独的类来创建图形对象的实现方式。5.4.2 中介绍的 graphics 库则采用了更符合面向对象概念的实现。

其中第一行的含义是在画布 c 上创建了一个矩形（稍后详述），create_rectangle()返 回的标识号被赋值给变量 r1，同时将该矩形与标签"#1"相关联。同样地，第二行创建了另 一个矩形，该矩形的标识号被赋值给变量 r2，该矩形还与两个标签"myRect"和"#2"相关 联。第三行的含义是将第一个矩形的标签重新设置为"myRect"和"rectOne"（注意原标签"#1"即告失效），这里使用了标识号 r1 来引用第一个矩形。第四行的含义是为具有标签 "rectOne"的图形项（即第一个矩形）添加一个新标签"ourRect"，这里使用了标签来引 用第一个矩形。至此，第一个矩形与 3 个标签"myRect"、"rectOne"和"ourRect"相关 联，其中任何一个都可用于引用该图形。注意，标签"myRect"同时引用两个矩形。

Canvas 还预定义了 ALL（或"all"）标签，此标签与画布上的所有图形项相关联。

画布对象的方法

上面例子中介绍了画布对象的 itemconfig()和 addtag_withtag()方法，除此之 外，画布对象还提供很多方法用于对画布上的图形项进行各种各样的操作。将图形项的标识 号或标签作为参数传递给画布对象的方法，即可指定被操作的图形项。下面再介绍几个画布 对象的常用方法。

gettags()方法可用于获取与给定图形项相关联的所有标签。例如下面的语句显示标 识号为 r1 的图形项的所有关联标签：

```
>>> print c.gettags(r1) ('myRect', 'rectOne', 'ourRect')
```

find_withtag()方法可用于获取与给定标签相关联的所有图形项。例如下面的语句显示与"myRect"标签相关联的所有图形项，返回结果为各图形项的标识号所构成的元组：

```
>>> print c.find_withtag("myRect") (1,2)
```

delete()方法用于从画布上删除指定的图形项。例如下面的语句从画布上删除第一个矩形：

```
>>> c.delete(r1)
```

move()方法用于在画布上移动指定图形。例如，为了将第二个矩形在 x 方向移动 10像素，在 y 方向移动 20 像素，可以执行下列语句：

```
>>> c.move(r2,10,20)
```

读者可查阅 Tkinter 资料以了解更多的画布对象方法。

5.2.3 在画布上绘图

本节介绍如何在画布上绘制图形。为了完整起见，我们将前面介绍过的首先需要执行的

几条语句合在一起复制如下：

```
>>> from Tkinter import *
>>> root = Tk()
>>> c = Canvas(root,width=300,height=200,bg='white')
>>> c.pack()
```

如前所述，c 是一个画布对象，而画布对象提供了若干方法用于绘制矩形、椭圆、多边 形等图形。为了画特定图形，只要向画布对象 c 发出执行特定方法的请求即可。

矩形

画布对象提供 create_rectangle()方法，用于在画布上创建矩形。矩形的位置和大 小由两点定义：(x0,y0)给出矩形的左上角，(x1,y1)给出矩形的右下角①。用法如下：

```
create_rectangle(x0,y0,x1,y1,<选项设置>...)
id = create_rectangle(x0,y0,x1,y1,<选项设置>...)
```

> ① 准确地说，矩形并不包含(x1,y1)，即(x1,y1)位于矩形右下角之外。例如，用左上角(10,10)和右下角(12,12)

定义的矩形实际上大小是 2 像素×2 像素，包含像素(11,11)但不包含像素(12,12)。

create_rectangle()的返回值是所创建的矩形的标识号，第一种用法没有保存这个 标识号，而第二种用法将标识号存入了一个变量。

例如，下面的语句创建一个以(50,50)为左上角、以(200,100)为右下角的矩形：

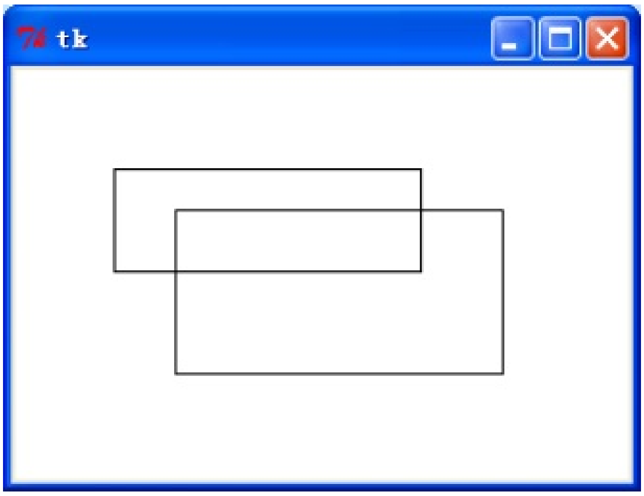
```
>>> c.create_rectangle(50,50,200,100) 1
```

结果如图 5.5(a)所示。注意，语句返回的 1 是矩形的标识号，表示这个矩形是画布上的 1 号

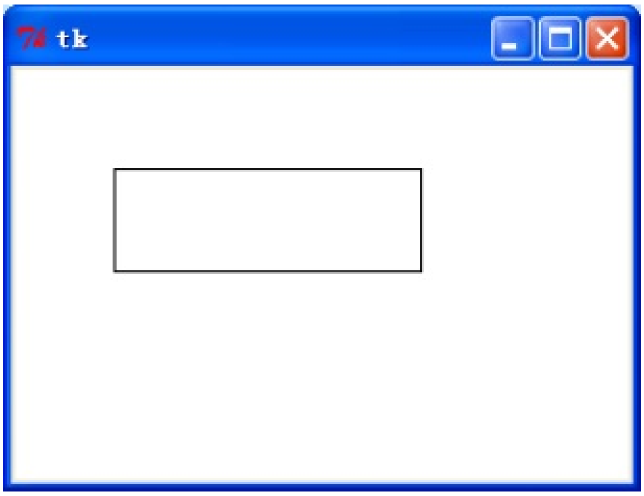
图形项。为了将来在程序中引用图形，最好用变量来保存图形的标识号，或者将图形与某个标签相关联。例如我们再创建一个矩形：

```
>>> r2 = c.create_rectangle(80,70,240,150,tags="rect#2")
>>> print r2 2
```

结果如图 5.5(b)所示。可见，第二个矩形的标识号为 2，它还与标签"rect#2"相关联。



(a)



(b)

图 5.5 在画布上画矩形

矩形图形实际上可视为由两个部分组成：轮廓线和内部。矩形轮廓线可以用选项 outline 来设置颜色，其缺省值是黑色，即等同于设置 outline="black"。如果将 outline 设置为空串""，则不显示轮廓线（透明的轮廓线）。轮廓线的宽度可以用选项 width 来设置，缺省值为 1 像素。矩形内部可以用选项 fill 来设置填充颜色，此选项的 缺省值是空串，即等同于设置 fill=""，效果是内部透明。

轮廓线可以画成虚线形式，这需要用到选项 dash，该选项的值是整数元组。最常用的 是 2 元组(a,b)，其中 a 指定要画几个像素，b 指定要跳过几个像素，依此重复，直至轮廓 线画完。若 a、b 相等，可以简记为(a,)。

矩形还有个选项 state，用于设置图形的显示状态。缺省值是 NORMAL（或"normal"）， 即正常显示。另一个有用的值是 HIDDEN（或"hidden"），它使矩形在画布上不可见。使 一个图形在 NORMAL 和 HIDDEN 两个状态之间交替变化，能形成闪烁的效果。

另外，5.2.2 中介绍过可以用选项 tags 为矩形关联标签，这里就不重复了。 上面例子中没有为两个矩形设置任何选项，即所有选项都取各自的缺省值。如 5.2.2 中所介绍的，我们可以利用画布对象的方法 itemconfig()来设置选项值。例如：

```
>>> c.itemconfig(1,fill="black")
>>> c.itemconfig(r2,fill="grey",outline="white",width=6)
```

执行后的结果如图 5.6 所示。

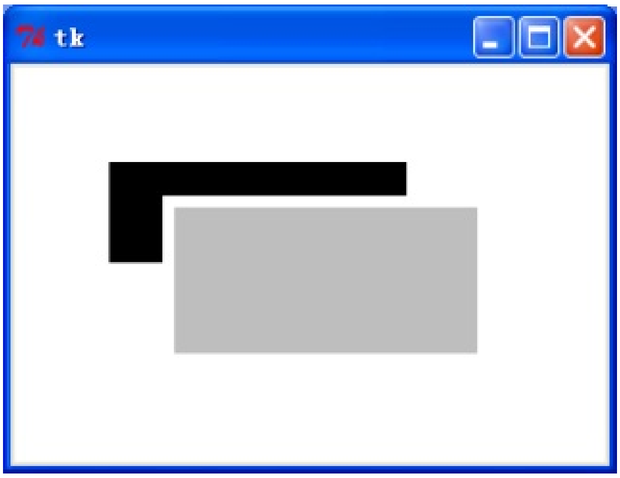
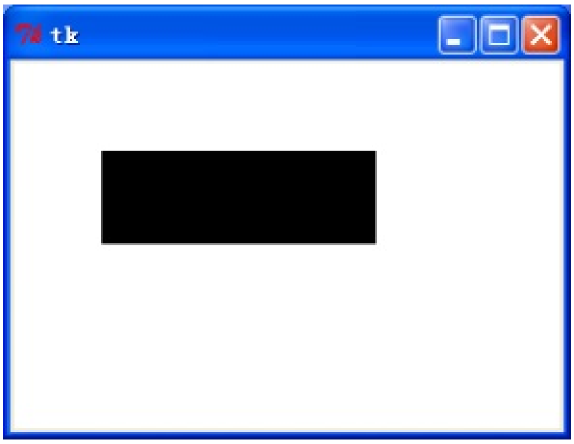


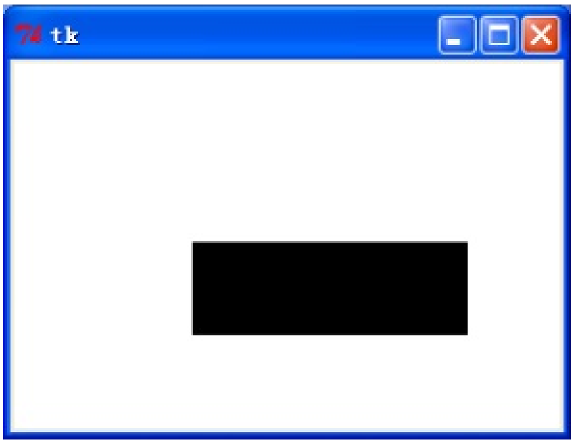
图 5.6 修改图形选项

将图 5.5 和图 5.6 相比较即可看出，在画布上，后创建的矩形是覆盖在先创建的矩形之 上的，并且未涂色时矩形内部是透明的（能看到被覆盖的矩形的轮廓线）。事实上，画布上 的所有图形项都是按创建次序堆叠起来的，第一个创建的图形项处于画布最底部（最靠近背景），最后创建的图形项处于画布最顶部（最靠近前景）①。图形的位置如果有重叠，则上面的图形会遮挡下面的图形。

如上一小节所介绍的，可以使用画布对象的 delete()和 move()方法删除和移动图 形。例如下列语句删去第二个矩形（结果见图 5.7(a)），并将第一个矩形在 x 轴和 y 轴方向 都移动 50 个像素（结果见图 5.7(b)）：

```
>>> c.delete(r2)
>>> c.move(1,50,50)
```





(b)

图 5.7 图形的删除和移动

至此我们详细介绍了矩形的画法、选项设置和图形操作（删除、移动等），接下去介绍 其他图形时会有很多相似的内容，以后我们将不重复详述。

顺便提一下，画布对象没有提供画“点”的方法，但我们可以画一个极小的矩形来当作 点。例如：

```
>>> c.create_rectangle(50,50,51,51)
```

> ① 即所有图形项形成一个显示列表。画布提供方法来对此列表重新排序。具体方法可参考 Tkinter 资料。

最后说明一下绘制矩形时如何提供坐标数据。create_rectangle()方法中坐标参数 的形式是很灵活的，既可以直接提供坐标值，也可以先将坐标数据存入变量，然后将该变量 传给该方法；既可以将所有坐标数据构成一个元组，也可以将它们组成多个元组。例如，create_rectangle()方法中的四个坐标参数既可以如上面例子那样作为四个值，也可以 定义成两个点（两个 2 元组）分别赋值给两个变量，还可以定义成一个 4 元组并赋值给一个 变量。形如：

```
>>> p1 = (10,10)
>>> p2 = (50,80)
>>> c.create_rectangle(p1,p2,tags="#3")
>>> xy = (100,110,200,220)
>>> c.create_rectangle(xy)
```

将坐标存储在变量中的做法是值得推荐的，因为这更便于在绘制多个图形时计算、安排 它们的相互位置。要强调的是，这里介绍的内容对所有图形（只要用到坐标参数）都是适用 的。

椭圆形

画布对象提供 create_oval()方法，用于在画布上画一个椭圆形（特例是圆形）。椭 圆的位置和尺寸通过其限定框（bounding box，即外接矩形）决定，而限定框由左上角坐标 (x0,y0)和右下角坐标(x1,y1)定义（参见图 5.8）。

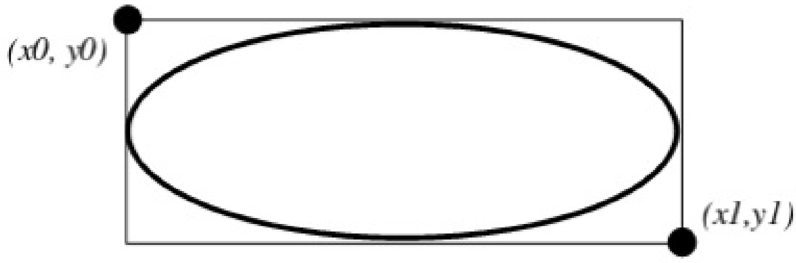


图 5.8 用限定框定义椭圆 创建椭圆的语句模板如下：

```
create_oval(x0,y0,x1,y1,<选项设置>...)
id = create_oval(x0,y0,x1,y1,<选项设置>...)
```

create_oval()的返回值是所创建的椭圆形的标识号，第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。

例如，下面的语句序列描绘地球绕太阳旋转的轨道，其中分别创建了一个椭圆形和两个 圆形，并且为大圆形涂上红色表示太阳，为小圆形涂上蓝色表示地球（参见图 5.9）。

```
>>> o1 = c.create_oval(50,50,250,150)
>>> o2 = c.create_oval(110,85,140,115,fill='red')
>>> o3 = c.create_oval(245,95,255,105,fill='blue')
```

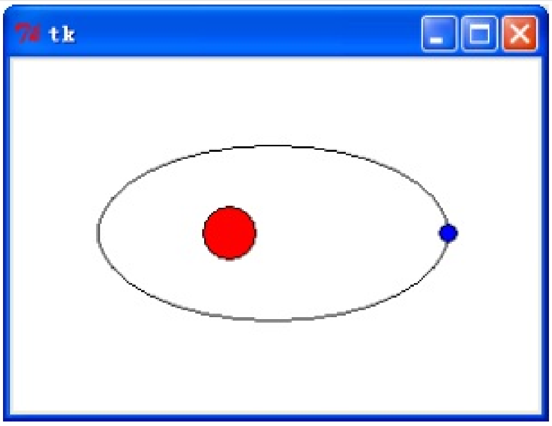


图 5.9 椭圆和圆

和矩形类似，椭圆形的常用选项包括 fill、outline、width、dash、state 和 tags等。

画布对象的 delete()方法、move()方法和 itemconfig()方法同样可用于椭圆形 的删除、移动和选项设置。

弧形

画布对象提供 create_arc()方法，用于在画布上创建一个弧形。与椭圆的绘制类似，create_arc()的参数是用来定义一个矩形的左上角和右下角的坐标，该矩形唯一确定了 一个内接椭圆形（特例是圆形），而最终要画的弧形是该椭圆的一段。创建弧形的语句模板 如下：

```
create_arc(x0,y0,x1,y1,<选项设置>...)
id = create_arc(x0,y0,x1,y1,<选项设置>...)
```

create_arc()的返回值是所创建的弧形的标识号，第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。

弧形的开始位置由选项 start 定义，其值为一个角度（以 x 轴方向为 0 度）；弧形的结 束位置由选项 extent 定义，其值表示从开始位置逆时针旋转的角度。start 的缺省值为0 度，extent 的缺省值为 90 度。显然，如果 start 设置为 0 度，extent 设置为 360 度， 则画出一个完整的椭圆形，效果和 create_oval()方法一样。

选项 style 用于规定弧形的式样，可以取三种值：PIESLICE 或"pieslice"是扇形（弧形两端与圆心相连），ARC 或"arc"是弧（圆周上的一段），CHORD 或"chord"是弓形（弧加连接弧两端点的弦）。style 的缺省值是 PIESLICE。如图 5.10 所示。

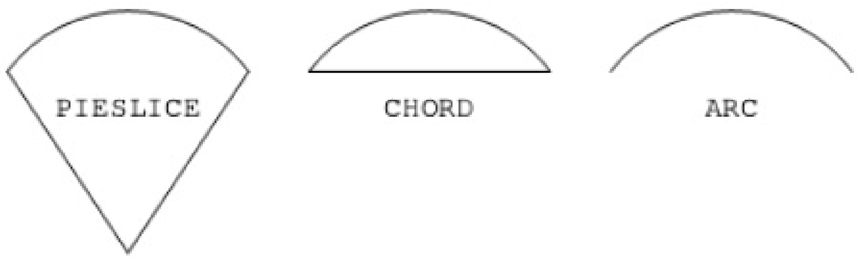


图 5.10 三种弧形

弧形的其他常用选项 outline、fill、width、dash、state 和 tags 的意义和缺 省值都和矩形类似。注意只有 PIESLICE 和 CHORD 形状才可填充颜色。

下面的例子画了一个扇形、一条弧和一个弓形，结果如图 5.11 所示。

```
>>> bbox = (50,50,250,150)
>>> c.create_arc(bbox)
>>> c.create_arc(bbox,start=100,extent=140,style="arc",width=4)
>>> c.create_arc(bbox,start=250,extent=110,style="chord")
```

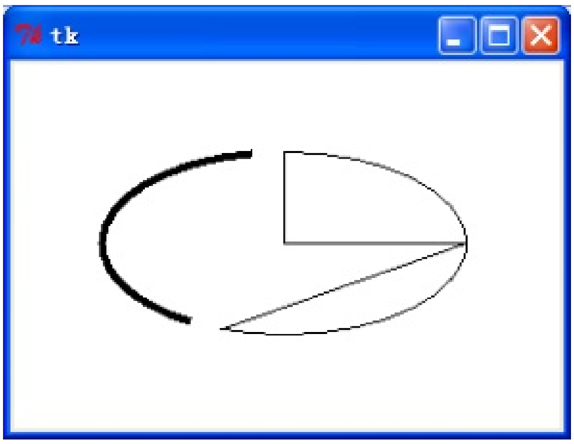


图 5.11 弧形

画布对象的 delete()方法、move()方法和 itemconfig()方法同样可用于弧形的 删除、移动和选项设置。

线条

画布对象提供 create_line()方法，用于在画布上创建连接多个点的线段序列，其语 句模板如下：

```
create_line(x0,y0,x1,y1,...,xn,yn,<选项设置>...)
id = create_line(x0,y0,x1,y1,...,xn,yn,<选项设置>...)
```

create_line()方法将各点(x0,y0)、(x1,y1)、…、(xn,yn)按顺序用线条连接起来，返回值是所创建的线条的标识号。第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。 没有特别说明的话，相邻两点间用直线连接，即图形整体上是条折线。但如果将选项 smooth 设置成非 0 值，则各点被解释成 B-样条曲线的顶点，图形整体是一条平滑的曲线。

不同于矩形、椭圆、弧形中的扇形和弓形的是，线条不能形成轮廓线和内部区域两部分，因此没有 outline 选项，只有 fill 选项。选项 fill 在此意为线条的颜色，其缺省值为 黑色。选项 width 设置线条宽度，缺省值为 1 像素。

线条可以通过选项 arrow 来设置箭头，该选项的缺省值是 NONE（无箭头）。如果将 arrow 设置为 FIRST 或"first"，则箭头在(x0,y0)端；设置为 LAST 或"last"，则箭 头在(xn,yn)端；设置为 BOTH 或"both"，则两端都有箭头。

选项 arrowshape 用于描述箭头形状，其值为 3 元组(d1,d2,d3)，含义如图 5.12 所 示。缺省值为(8,10,3)。

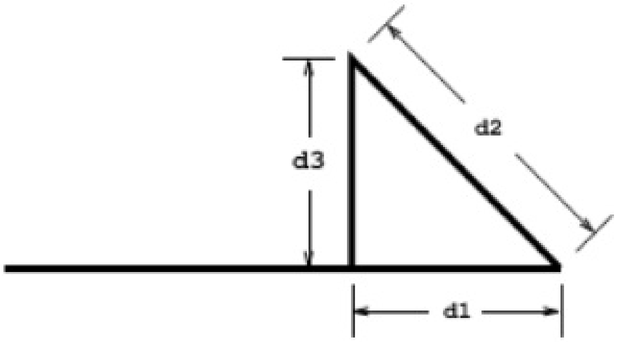


图 5.12 箭头形状的定义

线条和前面介绍的各种图形一样，具有 dash、state、tags 等选项。

下面的语句序列画的是北斗七星（大熊座）和北极星：其中 s1 到 s7 以及 polaris

给出了各星的坐标，我们在这些位置创建了涂黑的圆形表示北斗七星和北极星，然后用直线段依次连接 s1 到 s7，另外沿 s6－s7 延长线方向画了根带箭头的虚线指向北极星①。最下 方曲线表示地球表面，这里虽然只提供了三个点，但 Tkinter 仍能画出一条相当平滑的曲线。

```
>>> s1 = (20,20)
>>> s2 = (60,40)
>>> s3 = (80,60)
>>> s4 = (85,80)
>>> s5 = (70,100)
>>> s6 = (85,115)
>>> s7 = (110,100)
>>> polaris = (220,40)
>>> c.create_oval(s1,(23,23),fill='black')
>>> c.create_oval(s2,(63,43),fill='black')
>>> c.create_oval(s3,(83,63),fill='black')
>>> c.create_oval(s4,(88,83),fill='black')
>>> c.create_oval(s5,(73,103),fill='black')
>>> c.create_oval(s6,(88,118),fill='black')
>>> c.create_oval(s7,(113,103),fill='black')
>>> c.create_oval((222,36),(226,42),fill='black')
>>> c.create_line(s1,s2,s3,s4,s5,s6,s7,s4)
>>> c.create_line(s7,polaris,dash=(4,),arrow=LAST)
>>> c.create_line(5,190,150,160,295,190,smooth=1)
```

结果如图 5.13 所示。

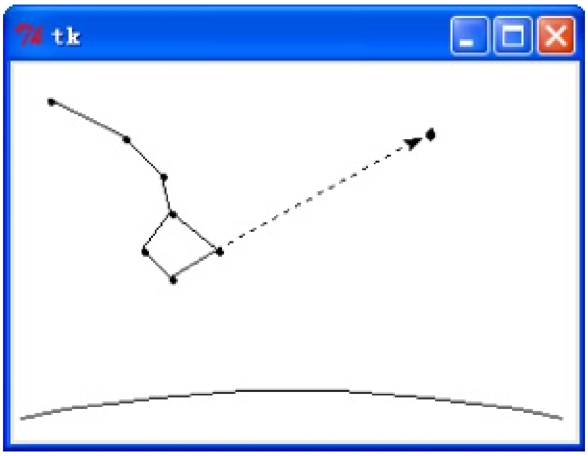


图 5.13 线条

画布对象的 delete()方法、move()方法和 itemconfig()方法同样可用于线条的 删除、移动和选项设置。

多边形

画布对象提供 create_polygon()方法，用于在画布上创建一个多边形。与线条类似， 多边形是用一系列顶点（至少三个）的坐标定义的，系统将把这些顶点按次序连接起来。与 线条不同的是，最后一个顶点需要与第一个顶点连接，从而形成封闭的形状。

> ① 这是利用北斗七星寻找北极星进行定向的常识方法。

创建多边形的语句模板如下：

```
create_polygon(x0,y0,x1,y1, ..., <选项设置>...)
id = create_polygon(x0,y0,x1,y1, ..., <选项设置>...)
```

create_polygon()的返回值是所创建多边形的标识号，第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。

和矩形类似，outline 和 fill 分别设置多边形的轮廓线颜色和内部填充色；但与矩 形不同的是，多边形的 outline 选项缺省值为空串，即轮廓线不可见，而 fill 选项的缺 省值为黑色。

与线条类似，一般用直线连接顶点，但如果将选项 smooth 设置成非 0 值，则表示用B-样条曲线连接顶点，这样绘制的是由平滑曲线围成的图形。

下面的语句序列以不同方式连接 5 个点，或设置不同的选项值，形成三种不同的五边形：

```
>>> p11,p21,p31 = (70,20),(70+100,20),(70,20+100)
>>> p12,p22,p32 = (35,50),(35+100,50),(35,50+100)
>>> p13,p23,p33 = (55,85),(55+100,85),(55,85+100)
>>> p14,p24,p34 = (85,85),(85+100,85),(85,85+100)
>>> p15,p25,p35 = (105,50),(105+100,50),(105,50+100)
>>> c.create_polygon(p11,p12,p13,p14,p15)
>>> c.create_polygon(p21,p23,p25,p22,p24,outline="black",fill="")
>>> c.create_polygon(p31,p32,p33,p34,p35,outline="black",fill="")
```

执行结果如图 5.14 所示。

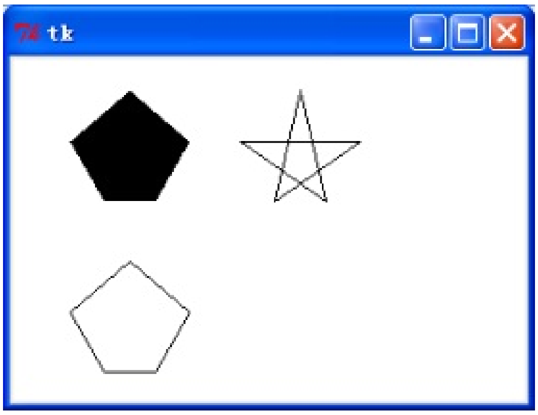


图 5.14 多边形

多边形的另几个常用选项 width、dash、state 和 tags 的用法都和矩形类似。 画布对象的 delete()方法、move()方法和 itemconfig()方法同样可用于多边形的删除、移动和选项设置。

文本

画布对象提供 create_text()方法，用于在画布上显示一行或多行文本。这里，文本 是作为图形对象看待的，与普通的字符串不同。创建文本的语句模板如下：

```
create_text(x,y,<选项设置>...)
id = create_text(x,y,<选项设置>...)
```

其中(x,y)指定显示文本的参考位置。create_text()的返回值是所创建的文本的标识 号，第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。

文本内容由选项 text 设置，其值就是显示的字符串。字符串中可以使用换行字符"\n"， 从而实现多行文本的显示。

选项 anchor 用于指定文本的哪个“锚点”与显示位置(x,y)对齐。首先想象文本有 个界限框，Tkinter 为界限框定义了若干个“锚点”，锚点用东南西北等方位常量表示，如图 5.15 所示。通过锚点可以控制文本的相对位置，例如，若将 anchor 设置为 SW，则将文本界限框的左下角置于参考点(x,y)；若将 anchor 设置为 N，则将文本界限框的顶边中点置 于参考点(x,y)。anchor 的缺省值为 CENTER，表示将文本的中心置于参考点(x,y)。

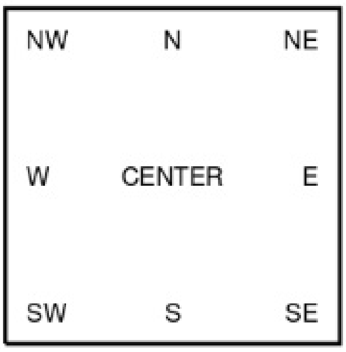
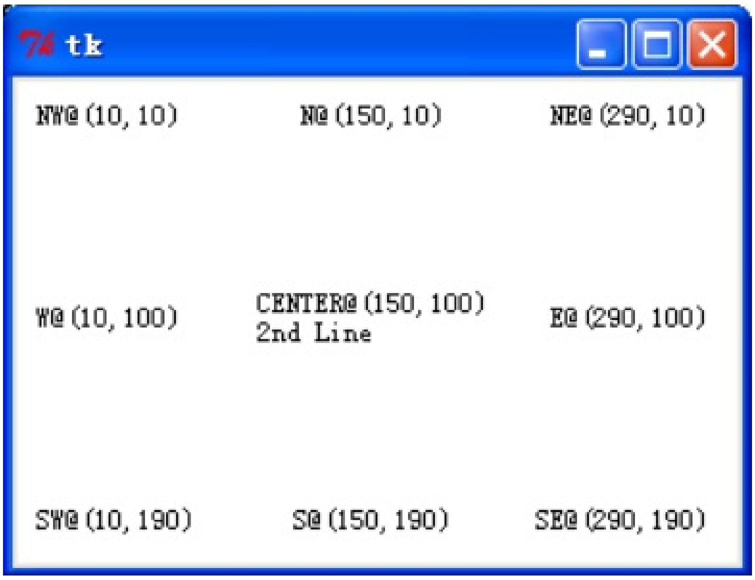


图 5.15 锚点

选项 fill 用于设置文本的颜色，缺省值为黑色。如果设置为空串，则文本不可见。 选项 justify 用于控制多行文本的对齐方式，其值为 LEFT、CENTER 或 RIGHT，缺省值为 LEFT。而 width 用于控制文本的宽度，超出宽度就要换行。 选项 state、tags 的意义同前。 下面的语句序列演示了如何在画布上安排文本：

```
>>> t1 = c.create_text(10,10,text="NW@(10,10)",anchor=NW)
>>> c.create_text(150,10,text="N@(150,10)",anchor=N)
>>> c.create_text(290,10,text="NE@(290,10)",anchor=NE)
>>> c.create_text(10,100,text="W@(10,100)",anchor=W)
>>> c.create_text(150,100,text="CENTER@(150,100)\n2nd Line")
>>> c.create_text(290,100,text="E@(290,100)",anchor=E)
>>> c.create_text(10,190,text="SW@(10,190)",anchor=SW)
>>> c.create_text(150,190,text="S@(150,190)",anchor=S)
>>> c.create_text(290,190,text="SE@(290,190)",anchor=SE)
```



结果如图 5.16 所示。

图 5.16 文本

程序中可能需要读取或修改文本的内容，画布对象的 itemcget()和 itemconfig()方法可用于此目的。例如：

```
>>> print c.itemcget(t1,"text")
NW@(10,10)
>>> c.itemconfig(t1,text="NorthWest")
```

其中第一行读取标识号为 t1 的文本项的 text 选项值；第三行将标识号为 t1 的文本的 text 选项重新设置为"NorthWest"。

画布对象的 delete()方法、move()方法同样可用于文本的删除和移动。

图像

除了在画布上自己画图，还可以将来自文件的现成图像显示在画布上。Tkinter 针对不 同格式的图像文件有不同的显示方法，这里我们只介绍显示 gif 格式图像的方法。

第一步是利用 Tkinter 提供的 PhotoImage 类来创建图像对象，语句模板如下：

```
img = PhotoImage(file = <图像文件名>)
```

其中选项 file 用于指定图像文件（支持 gif、pgm、ppm 格式①），PhotoImage()返回值是一个图像对象，这里我们用变量 img 引用这个对象，接下去将把这个图像对象显示在画布中②。

第二步是在画布上显示图像，这可通过画布对象提供的 create_image()方法完成。 该方法的用法如下：

```
c.create_image(x,y,image = <图像对象>,<选项设置>...)
id = c.create_image(x,y,image=<图像对象>,<选项设置>...)
```

其中(x,y)是决定图像显示位置的参考点；image 选项决定显示的图像，其值就是第一步创建的图像对象。create_image()的返回值是所创建的图像在画布上的标识号，第一种用法没有保存这个标识号，而第二种用法将标识号存入了一个变量。

图像在画布上的位置由参考点(x,y)和 anchor 选项决定，具体设置与文本相同。 例如，假设电脑上有个文件 C:\WINDOWS\Web\exclam.gif，则下列语句序列首先为该图像文件创建了一个图像对象 pic，然后将该图像对象显示在了画布中：

```
>>> pic = PhotoImage(file = "C:\WINDOWS\Web\exclam.gif")
>>> c.create_image(150,100,image=pic)
```



结果如图 5.17 所示。

图 5.17 画布上显示图像 可以为图像设置选项 state、tags，意义同前。

画布对象的 delete()方法、move()方法同样可用于图像的删除和移动。

> ① 至于常见的 jpg 格式或其他格式的图像，可以利用 Python 图像库（PIL）转换成 Tkinter 图像对象。
>
> ② 还可以显示在按钮（Button）、标签（Label）、文本（Text）等 GUI 构件中，参见第 8 章。

除了以上各种图形、文字和图像，我们还可以在画布上放置其他 GUI 构件，例如按钮、勾选钮等，以便用户更好地与画布进行交互。读者学习了第 8 章后可以试着编写这样的程序。

5.2.4 图形的事件处理

面向对象的概念是和事件驱动编程联系在一起的。所谓事件是指在程序执行过程中发生的事情，例如点击了鼠标左键、按下了键盘上的回车键之类。某个对象可以与特定事件绑定 在一起，这样当特定事件发生时，可以调用特定的函数来处理这个事件。

画布及画布上的图形都是对象，都可以与交互事件绑定，这样用户可以利用键盘、鼠标 来操作、控制画布和图形。第 8 章将详细介绍 Tkinter 的事件驱动编程，这里我们只用一个 简单例子展示画布和图形对象的事件处理能力。

【程序 5.1】eg5_1.py

```
from Tkinter import *
def canvasFunc(event):
    if c.itemcget(t,"text") == "Hello!":
        c.itemconfig(t,text="Goodbye!")
    else:
        c.itemconfig(t,text="Hello!")
def textFunc(event):
    if c.itemcget(t,"fill") != "white":
        c.itemconfig(t,fill="white")
    else:
        c.itemconfig(t,fill="black")
root = Tk()
c = Canvas(root,width=300,height=200,bg='white')
c.pack()
t = c.create_text(150,100,text="Hello!")
c.bind("<Button-1>",canvasFunc)
c.tag_bind(t,"<Button-3>",textFunc)
root.mainloop()
```

下面我们对此程序中与事件处理有关的几个要点进行说明。

事件绑定

对象 O 需要与特定事件 E 进行绑定，以便告诉系统当针对 O 发生了 E 之后该如何处理。 程序 5.1 的倒数第 3 行中，利用画布的 bind()方法将画布对象 c 与鼠标左键点击事件"<Button-1>"进行了绑定，其中告诉系统当用户在画布 c 上点击鼠标左键时，就去执行 函数 canvasFunc()。

程序 5.1 的倒数第 2 行中，利用画布的 tag_bind()方法将画布对象 c 上的图形项（文 本）t 与鼠标右键点击事件"<Button-3>"进行了绑定，其中告诉系统当用户在文本 t 上 点击鼠标右键时，就去执行函数 textFunc()。

事件处理函数

程序员可以自定义对事件的处理函数。

程序 5.1 中定义了 canvasFunc()函数用于处理画布上的鼠标左键点击事件，其功能 是改变文本 t 的内容：如果当前内容是"Hello!"就改成"Goodbye!"，如果当前是 "Goodbye!"就改成"Hello!"。每当用户在画布上点击鼠标左键时就执行一次这个函数， 形成文字内容随鼠标点

击而切换的效果。

程序 5.1 中还定义了 textFunc()函数用于处理文本上的鼠标右键点击事件，其功能是 改变文本 txt 的颜色：如果当前不是白色则改为白色，否则改为黑色。每当用户在文本上 点击鼠标右键时就执行一次这个函数，形成文本随鼠标点击而出没的效果。注意画布背景色 是白色，因此将文本设置为白色就相当于隐去文本。

主事件循环

程序 5.1 中并没有调用上述两个事件处理函数的语句，它们是由系统根据所发生的事件 而 自动调用 的 。系统 如 何知道 现 在发生 了 什么事 件 呢？程 序 5.1 中最后一 行 root.mainloop()的意义是进入根窗口的主事件循环。执行了这一条语句之后，系统就会 自行监控在根窗口上发生的各种事件，并触发相应的处理函数。

以上对 Tkinter 的事件处理作了简单介绍，如果读者仍有疑惑，第 8 章中有详细介绍。

5.3 编程案例

5.3.1 统计图表

图形的一个重要用途是为数据提供可视的表示，这在统计、汇总性质的应用程序中尤其 重要，因为汇总数据几乎都可以利用图形来改善表示。下面我们编写一个简单的统计汇总程序，以演示图形编程在数据可视化方面的应用。

假设某高校的老师在考试后需要根据学生的考试成绩来分析试卷，以判断试卷是偏难、 偏容易还是适中。难度适中的试卷应该导致正态分布的成绩。为帮助老师完成试卷分析，我 们编写一个统计汇总程序，其功能是：老师输入考试分数（百分制），然后程序将分数换算 成等级制（分为 A、B、C、D、F 五等）并统计各等级分数的人数，最后画一个饼图来直观 地给出各等级人数的比例。

程序规格

输入：考试分数。

输出：以饼图表示的各分数段所占比例。

算法设计

本程序在算法上很简单，属于典型的 IPO（输入－处理－输出）模式。不过虽然算法很 简单，但是在绘制图形方面需要花费大量精力，因为绘图涉及精确的坐标、形状、颜色等细节，还需要整个图形画面看上去整齐、匀称、美观。可以说，图形编程中大量时间都花在了这类“美工”任务之上。

首先，由用户输入每个学生的分数（百分制）。然后根据该分数所对应的等级去累加各 等级人数变量。输入结束后，总人数和各等级人数就确定了。

其次，计算各分数等级人数占总人数的比例。

然后，根据比例绘制饼图。在 Tkinter 编程中，这需要先创建窗口和画布，然后利用画 布的 create_arc()方法绘制代表五个等级的五个扇形。扇形的角度反映了各分数等级的

比例，扇形具有不同填充色以相互区分。为了显示各扇形对应的等级，还需要绘制图例。 最后，用户通过饼图各扇形的大小只能看出各分数等级所占的大致比例。精确的比例值

当然可以固定显示在画面中，不过我们采用另一种更有趣的设计：当用户将鼠标指针移入某个扇形中时，画布上就显示该扇形所代表的比例值。

以上步骤还需要进一步明确细节，最主要的就是窗口、画布的大小和各图形项的精确位 置等。通过用草图等手段做一些计算和试验，最终确定如图 5.18 所示的设计：

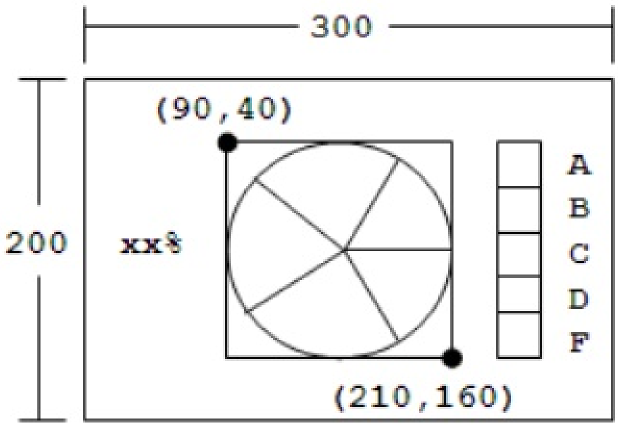


图 5.18 画布图形项设计

至此，可以写出本程序的算法伪代码。

```
算法：
用户输入考试分数 mark，并根据 mark 对应的等级累加各等级的人数 a、b、c、d、f；
创建窗口和大小为 300x200 的画布；
计算各分数等级的比例（a/n 等），并据此确定每个扇形的起止角度（sA、eA 等）；
绘制各个扇形；
绘制图例； 为各扇形绑定“鼠标进入”事件，并定义事件处理函数（inPieA()等）；
进入主事件循环。
```

代码实现

从上面的算法很容易翻译成 Python 代码。程序 5.2 中所用到的知识都在前面介绍过， 只有“鼠标进入”事件的处理需要说明一下。

当鼠标指针移到某个图形项上面时即发生事件"<Enter>"，这时系统触发所绑定的事 件处理函数（如 inPieA），这些函数的功能是计算该图形项对应的比例值，然后显示在画 布上的指定位置。另外由于事件处理函数中需要引用画布对象和各图形项，所以我们将这些 函数的定义放在了 main()函数内部，以便它们能引用 main()中定义的变量，即 cv、 piepct、a、b、c、d、f 和 n。

【程序 5.2】piechart.py

```
from Tkinter import *
def getMarks():
    a,b,c,d,f = 0,0,0,0,0
    mark = input("Enter a mark: ")
    while mark >= 0:
        if mark >= 90:
            a = a + 1
        elif mark &gt;= 80:
            b = b + 1
        elif mark &gt;= 70:
            c = c + 1
        elif mark &gt;= 60:
            d = d + 1
        else:
            f = f + 1
        mark = input("Enter a mark: ")
    return a,b,c,d,f
def main():
    a,b,c,d,f = getMarks()
    win = Tk()
    cv = Canvas(win,width=300,height=200,bg="white")
    cv.pack()
    n = a+b+c+d+f
    eA,sA = 360.0*a/n,0
    eB,sB = 360.0*b/n,eA
    eC,sC = 360.0*c/n,eA+eB
    eD,sD = 360.0*d/n,eA+eB+eC
    eF,sF = 360.0*f/n,eA+eB+eC+eD
    bb = (90,40,210,160)
    pieA = cv.create_arc(bb,start=sA,extent=eA,fill="yellow")
    pieB = cv.create_arc(bb,start=sB,extent=eB,fill="green")
    pieC = cv.create_arc(bb,start=sC,extent=eC,fill="black")
    pieD = cv.create_arc(bb,start=sD,extent=eD,fill="gray")
    pieF = cv.create_arc(bb,start=sF,extent=eF,fill="red")
    cv.create_rectangle(240,40,260,50,fill="yellow") cv.create_rectangle(240,40+24,260,50
    cv.create_text(270,40,text="A",anchor=N)
    cv.create_text(270,40+24,text="B",anchor=N) cv.create_text(270,40+48,text="C",anchor=
    piepct = cv.create_text(40,100,text="")
    def inPieA(event):
        pct = "%5.1f%%" % (100.0*a/n)
        cv.itemconfig(piepct,text=pct)
    def inPieB(event):
        pct = "%5.1f%%" % (100.0*b/n)
        cv.itemconfig(piepct,text=pct)
    def inPieC(event):
        pct = "%5.1f%%" % (100.0*c/n)
        cv.itemconfig(piepct,text=pct)
    def inPieD(event):
        pct = "%5.1f%%" % (100.0*d/n)
        cv.itemconfig(piepct,text=pct)
    def inPieF(event):
        pct = "%5.1f%%" % (100.0*f/n)
        cv.itemconfig(piepct,text=pct)
    cv.tag_bind(pieA,"&lt;Enter&gt;",inPieA)
    cv.tag_bind(pieB,"&lt;Enter&gt;",inPieB)
    cv.tag_bind(pieC,"&lt;Enter&gt;",inPieC)
    cv.tag_bind(pieD,"&lt;Enter&gt;",inPieD)
    cv.tag_bind(pieF,"&lt;Enter&gt;",inPieF)
    win.mainloop()
main()
```

程序 5.2 的一次运行结果如图 5.19 所示。

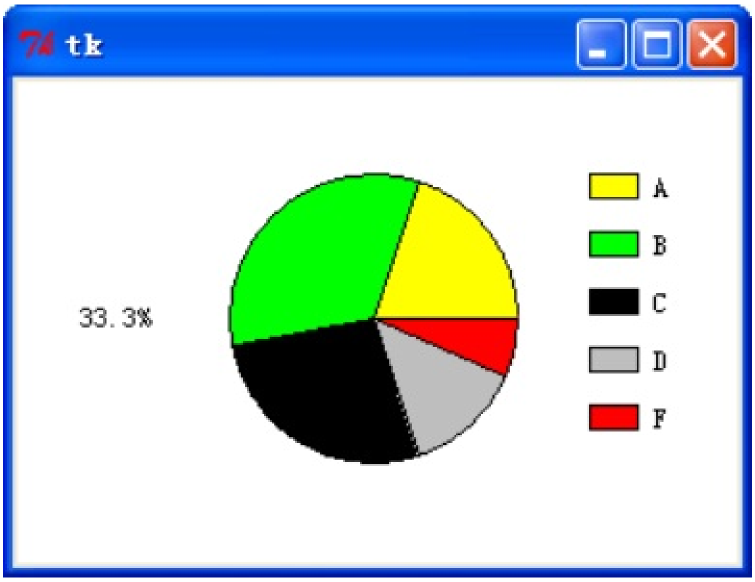


图 5.19 程序 5.2 的一次执行结果

5.3.2 计算机动画

顾名思义，动画就是运动的画面，计算机动画就是通过计算机编程来实现运动的画面。计算机动画在很多领域中都有应用，例如游戏开发、电影电视制作、教学演示等。计算机动 画并不神秘，只要掌握了静止图形的绘制方法，就很容易学会活动画面的制作。

现实世界中运动是连续的，而数字计算机只能处理离散量，因此计算机动画本质上只能 是对连续运动的近似和模拟。具体来说，动画是通过在屏幕上快速地交替显示一组静止图形（图像），或者让一幅图形（图像）快速地移动而实现的。每一幅静止图形（图像）称为一 帧，帧与帧之间在画面上只有小部分的不同，于是人眼的视觉暂留现象会使我们产生运动的 感觉。实验表明，每秒显示 24 帧画面能使人眼感觉到最佳的连续运动效果，所以在连续两帧画面之间应该停顿 0.04 秒左右。 动画制作有很多现成软件工具可用，例如在网页和多媒体教学中常用的 Flash。而我们在此介绍的是直接编程实现动画。

下面我们利用 Tkinter 来实现一个简单的动画程序。程序的功能是演示太阳、地球和月 球三个天体之间的运动情况，即月球绕地球运动，并且和地球一起绕太阳运动。

程序规格

输入：没有输入。

输出：演示太阳、地球和月球之间相对运动的动画。

算法设计

首先当然是建立窗口和画布，然后画出太阳、地球和月球三个天体，具体做法与 5.2.3 节中绘制椭圆的例子相似（图 5.9），当然现在需要多画一个月球，并且需要移动地球和月球。 本程序最关键的部分是解决地球和月球沿椭圆轨道移动的计算（假设太阳位置固定不动），下面先解决地球的运动问题。中学数学告诉我们，椭圆可以用如下方程来刻划：

```
x = a cos t
y = b sin t
```

因此，地球在轨道上自西向东旋转时的每一个位置(x,y)都可利用此方程算出，其中椭圆轨 道的 a、b 值是固定不变的，位置只由旋转角度 t 决定（参见图 5.20）。假设地球每次旋转 0.01p 弧度（这就是连续运动的离散化！），则地球的下一位置就是

```
x' = a cos(t + 0.01 pi)
y' = b sin(t + 0.01 pi)
```

由此可算出

```
dx = x' - x
dy = y' - y
```

于是可利用画布对象的 move()方法来移动地球到新的位置。

再看图 5.20，由于画布的坐标系原点不是椭圆轨道的中心，椭圆中心在画布坐标系中是(150,100)，故地球在 t 角度时的位置应该做个变换：

```
x = 150 + a cos t
y = 100 - b sin t
```

注意画布坐标系中 y 轴是向下的，因此上式计算 x 和 y 坐标时有加减的不同。

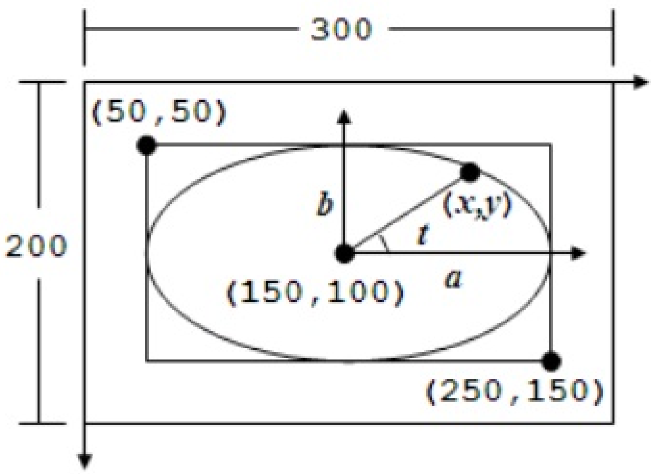


图 5.20 地球沿椭圆轨道旋转位置的计算

接下来解决月球的运动问题。首先月球是与地球一起沿椭圆轨道绕太阳运动的，因此月球相对于太阳的位置变化与地球一样由上述 dx 和 dy 决定，即程序 5.3 中的 edx 和 edy。 此外，月球又在绕地球旋转，利用同样方法可算出月球沿绕地球椭圆轨道（设 a、b 的值分 别为 20 和 15）运动时相对于地球的位置变化，即程序 5.3 中的 mdx 和 mdy。最终月球的位 置变化为 edx+mdx 和 edy+mdy。注意，月球绕地球的旋转速度大约是地球绕太阳的旋转 速度的 12 倍（一年有十二个月）。

解决了关键的地球月球的位置计算问题，本程序的算法就明确了，伪代码如下：

```
算法：
创建窗口和画布；
在画布上绘制太阳、地球和月球，以及地球的绕日椭圆轨道；
设置地球和月球的当前位置；
进入动画循环：
      旋转 0.01p；
      计算地球和月球的新位置；
      移动地球和月球到新位置
      更新地球和月球的当前位置；
      停顿一会
```

代码实现

上面的算法很容易翻译成如程序 5.3 所示的 Python 代码。代码中有两处需要说明一下： 第一，每次循环中修改图形位置后都必须执行一个更新画布显示的方法 c.update()，以 使新画面显示出来；第二，两个画面之间的停顿可以用 time 模块中的 sleep()函数来实 现，该函数的作用就是让程序休眠一会（参数以秒为单位）①。

【程序 5.3】animation.py

```
from Tkinter import *
from math import sin,cos,pi from time import sleep
def main():
    root = Tk()
    c = Canvas(root,width=300,height=200,bg='white')
    c.pack()
    orbit = c.create_oval(50,50,250,150)
    sun = c.create_oval(110,85,140,115,fill='red')
    earth = c.create_oval(245,95,255,105,fill='blue')
    moon = c.create_oval(265,98,270,103)
    eX = 250 # earth's X
    eY = 100 # earth's Y
    m2eX = 20 # moon's X relative to earth
    m2eY = 0 # moon's Y relative to earth
    t = 0
    while True:
        t = t + 0.01*pi
        new_eX = 150 + 100 * cos(t)
        new_eY = 100 - 50 * sin(t)
        new_m2eX = 20 * cos(12*t)
        new_m2eY = -15 * sin(12*t)
        edx = new_eX - eX
        edy = new_eY - eY
        mdx = new_m2eX - m2eX
        mdy = new_m2eY - m2eY
        c.move(earth,edx,edy)
        c.move(moon,mdx+edx,mdy+edy)
        c.update()
        eX = new_eX
        eY = new_eY
        m2eX = new_m2eX
        m2eY = new_m2eY
        sleep(0.04)
main()
```

> ① 如果不知道这个 sleep 函数，也可以自己写一个纯粹消磨时间的循环语句，例如循环 1 百万次，每次 都执行无用语句。同样能起到让画面停顿的效果。

图 5.21 是程序 5.3 执行过程中的一个屏幕截图。

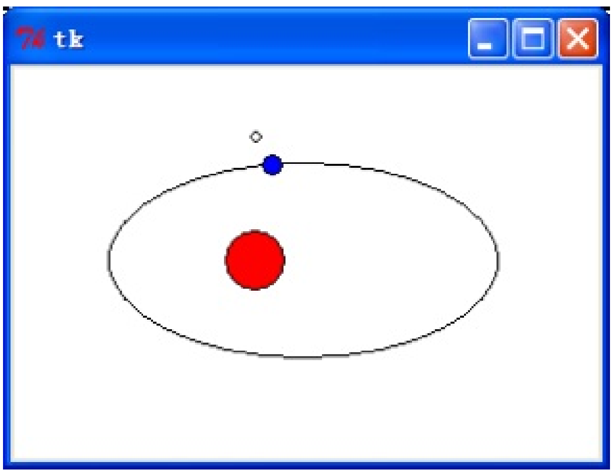


图 5.21 程序 5.3 的屏幕截图

5.4 软件的层次化设计：一个案例

一个复杂软件通常是由很多构件组成的，各构件之间的交互关系有多种模式。例如，在

面向过程编程中，一个程序通常是由多个子程序（过程或函数）组成的，各子程序之间通过调用和返回来进行交互。又如，在面向对象编程中，一个程序是由许多对象组成的，对象之间通过发送消息来进行交互。本节中我们通过案例来简单介绍一种常用的软件设计方法——层次化设计。

5.4.1 层次化体系结构

层次化设计是构造复杂系统的一个基本方法，按此方法设计出的系统具有层次化体系结构。现实世界中这种层次化结构俯拾皆是。例如，一幢高楼总是从最底层打基础开始，一层 一层地加高。又如，我国的行政组织具有街道、区、市、省、中央这样的层次化结构。

计算机软件的各个构件也经常组织成这样的层次体系结构。在层次体系中，下层构件为 上层构件提供服务，上层构件使用下层构件的服务，上层和下层之间形成一种类似“调用－ 返回”的关系。为了正确地调用和返回，每一层都需要提供一个界面（接口）给上层，以便 与之交互。层次体系顶层为程序员或最终用户提供界面。

我们在自顶向下逐步求精设计方法中也使用了层次化的设计，只不过那里的层次体现的 是功能的分解，即一个函数用更加细化的函数来实现，上下层之间就是函数的调用－返回关 系。而在这里讨论的是用于不同目的的层次化体系结构，其中上下层之间并非功能分解的关 系，分层是为了建立不同的界面。打个比方，假设有一种多功能电视机，其面板上有许多功 能按钮，然而多数老年人既不明白也不需要使用那些先进的功能，复杂的面板只会让老人连 简单的频道和音量按钮也搞不清。这时我们可以在原面板上覆盖一层新面板，其上只留下频 道和音量按钮，现在老人看到的电视机就有了简单易用的界面（图 5.22）。

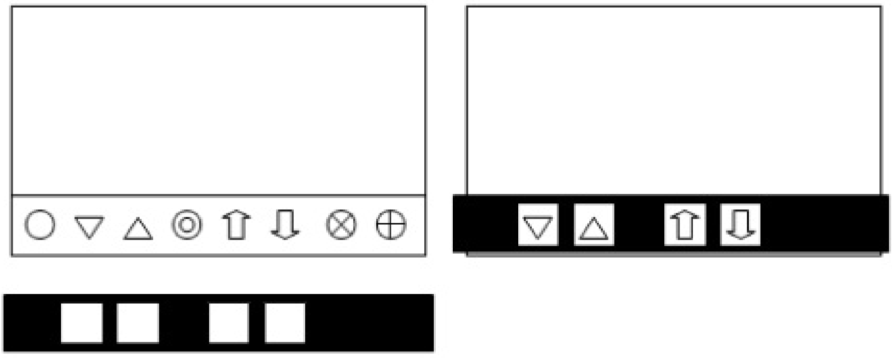


图 5.22 为电视机加一层面板

采用层次化设计的计算机软件的构件分成若干层，先有低层构件，然后在其上架设高层 构件。高层构件的功能依赖于低层构件的功能，但高层构件一般更容易理解，程序员或用户 使用起来更方便。典型的层次化软件体系结构的例子包括数据库的 ANSI-SPARC 三层模式、 网络技术的 ISO/OSI 七层模型、Web 应用开发中的三层体系结构等等。

层次化体系结构的主要优点包括重用和标准化。重用是指同样的构件可以用在任何具有 相同界面要求的地方；同样，只要层次间界面不变，一个构件也可以换用以不同方式实现的 其他同类构件。还以图 5.22 例打比方，我们自制的面板可以用于同品牌型号的所有电视机， 并且木质的面板可以换用塑料面板，黑色面板可以换用彩色面板，等等。标准化是指由标准 化组织为某一类软件构件定义标准界面，而各软件厂商可以采取不同的低层实现技术来实现 高层的标准界面。就好比家电协会规定所有电视机的面板都必须包括电源开关，而各厂商可 以用按钮来实现电源开关，也可以使用红外遥控来开关。

层次化体系结构的主要缺点是效率不如整体式结构，这是因为当程序员或用户面对顶层 构件请求某项服务时，这个请求需要从高层到低层逐层下传，最终由底层构件来实现功能， 再将结果逐层上传，直至顶层用户。这个逐层转换的过程显然很耗费时间。假如用户能直接 与底层打交道，功能的实现就会高效的多。

5.4.2 案例：图形库 graphics

如前所述，Tkinter 是 Python 语言的标准库，可以利用 Tkinter 中的画布构件来绘制图形。 虽然利用 Tkinter 来进行图形编程已经比较简单、方便，但对初学者来说可能还是有点小麻 烦。例如，画布甚至都没有提供画"点"的方法，初学者希望画点时往往不知怎么办。又如， 圆形一般都是通过圆心和半径来定义的，但在画布上画圆形时必须利用界限框（外接正方形） 来定义。另外，对图形的各种操作（如移动图形、修改图形的选项值等）都是通过调用画布 的方法来执行的，而根据面向对象的思想，更容易理解的做法应该是直接针对图形对象发出 操作请求。

由于上述理由，有人①在 Tkinter 之上写了一个更容易使用的图形库——graphics。这个 图形库是为教学目的而开发的，它将 Tkinter 的绘图功能以面向对象的方式重新包装了一下， 使得初学者更容易学习和应用。使用 graphics 提供的功能实际上就是使用 Tkinter 的功能， 但使用者并不知道这一点，也不需要知道这一点，这就是层次体系结构带来的效果。图 5.23 显示了 graphics 与 Tkinter 之间的关系，其中提到的 graphics 定义的各种图形类将在稍后介 绍。

> ① Python Programming: An Introduction to Computer Science 的作者 John Zelle。

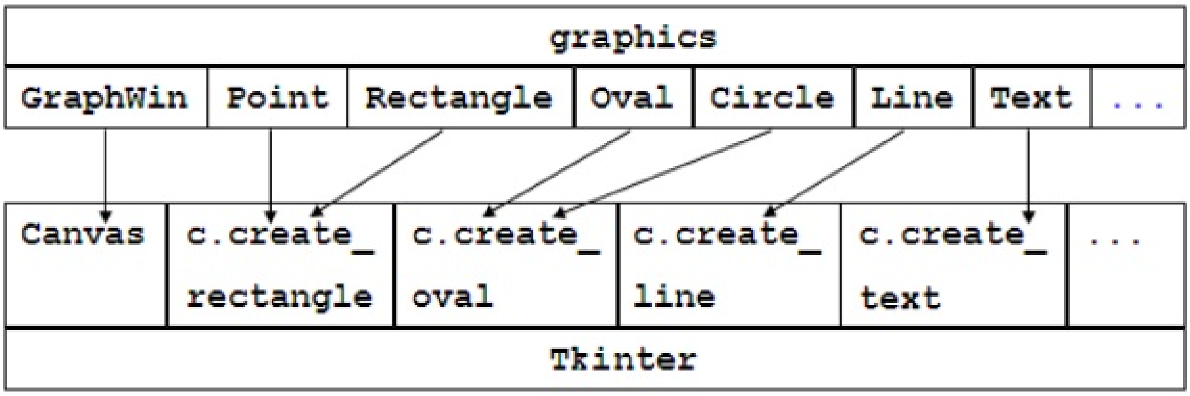


图 5.23 在 Tkinter 之上开发的 graphics

graphics 模块和说明文档可以从下列网站下载：

http://mcsp.wartburg.edu/zelle/python

下载后将 graphics.py 模块与你的图形程序放在一个目录中，或者放在 Python 安装目录 中即可。下面我们简要介绍如何使用 graphics 模块。

首先，需要导入 graphics 模块：

```
>>> from graphics import *
```

其次，创建一个绘图窗口：

```
>>> win = GraphWin("My Graphics Window",300,200)
```

这条语句的含义是在屏幕上创建一个窗口对象，窗口标题为"My Graphics Window"，宽 度为 300 像素，高度为 200 像素。三个参数都可以省略，缺省宽度和高度都是 200 像素。窗 口的坐标系仍然是我们熟悉的，即以窗口左上角为原点，x 轴向右，y 轴向下。

通过 Graphwin 类创建绘图窗口的界面实际上是对底层 Tkinter 中创建画布对象界面的重 新包装，也就是说，当程序员利用 graphics 模块创建绘图窗口时，系统会把这个请求向下转 达给 Tkinter 模块，而 Tkinter 模块就创建一个画布对象并返回给上层的 graphics 模块。这样 做不是没事找事多此一举，而是为了改善图形编程界面的易用性、易理解性。

接下去就可以在作图窗口中绘制图形了，稍后将介绍各种图形对象的创建方法。程序结 束后应该关闭图形窗口，为此只需向窗口对象发如下消息即可：

```
>>> win.close()
```

下面介绍 graphics 模块支持的各种图形对象的用法。演示代码中总是假定已经导入了 graphics 模块并创建了绘图窗口 win。

点

graphics 模块提供了类型 Point 用于在窗口中画点。创建点对象的语句模式为：

```
>>> p = Point(<x 坐标>,<y 坐标>)
```

下面通过一个交互过程来在窗口中创建 Point 对象，并演示 Point 对象的方法的使用。

```
>>> p = Point(100,80)
>>> p.draw(win)
>>> print p.getX(),p.getY() 100 80
>>> p.move(20,30)
>>> print p.getX(),p.getY() 120 110
```

第一条语句创建了一个 Point 对象，该点的坐标为(100,80)，变量 p 被赋值为该对象。

这时在窗口中并没有显示这个点，因为还需要让这个点在窗口中画出来，为此只需向对象 p 发送消息 draw()，这就是第二条语句的目的，其意为“请求对象 p 执行 draw(win)方法， 即在窗口 win 中将自己画出来”。第三条语句演示了 Point 对象的另两个方法 getX()和 getY()的使用，分别是获得点的 x 坐标和 y 坐标。第四条语句的含义是请求 Point 对象 p 改变位置，向 x 方向移动 20 像素，向 y 方向移动 30 像素。

此外，Point 对象还提供以下方法：

- p.setFill()：设置点 p 的颜色。
- p.setOutline：设置轮廓线的颜色。对 Point 来说，与 setFill 没有区别。

- p.undraw()：隐藏对象 p，即在窗口中变成不可见的。注意，隐藏并非删除，对象 p 仍然存在，随时可以重新执行 draw()。
- p.clone()：复制。复制一个与 p 一模一样的对象。

读者一定会觉得通过 Point 类来画点非常容易，但也会奇怪：graphics 是建立在 Tkinter 之上的一层软件，graphics 的所有功能都是依赖于 Tkinter 的功能实现的，但是 Tkinter 中并未提供画点功能啊。对这个疑问的解答很简单：Point 对象其实是 Tkinter 中的一个很小的矩形（参见图 5.23）！这是通过层次化改善图形编程界面的一个典型例子——当我们要画点时，就直接创建 Point 对象，而不是像在 Tkinter 中那样很别扭地创建一个矩形。

接下去介绍的其他图形对象就不再像 Point 这样详细解释并演示用法了，希望使用 graphics 模块的读者可以自行练习。

直线

直线类型为 Line，创建直线对象的语句模式为：

```
>>> line = Line(&lt;端点 1&gt;,&lt;端点 2&gt;)
```

其中两个端点都是 Point 对象。

和 Point 一样，Line 对象也支持 draw()、undraw()、move()、setFill()、setOutline()、clone() 等方法。此外，Line 对象还支持 setArrow()方法，用于为直线画箭头。

圆形

圆形类型为 Circle，创建圆形对象的语句模式为：

```
>>> c = Circle(&lt;圆心&gt;,&lt;半径&gt;)
```

其中圆心是 Point 对象，半径是个数值。

Circle 对象同样支持 draw()、undraw()、move()、setFill()、setOutline()、clone()等方法。此外，Circle 对象还支持 c.getRadius()方法，用于获取圆形对象 c的半径。

椭圆

椭圆类型为 Oval，创建椭圆对象的语句模式为：

```
>>> o = Oval(&lt;左上角&gt;,&lt;右下角&gt;)
```

其中左上角和右下角是两个 Point 对象，用于指定一个矩形，再由这个矩形定义一个内接椭圆。

椭圆对象同样支持 draw()、undraw()、move()、setFill()、setOutline()、 clone()等方法。

矩形

矩形类型为 Rectangle，创建矩形对象的语句模式为：

```
>>> r = Rectangle(&lt;左上角&gt;,&lt;右下角&gt;)
```

其中左上角和右下角是两个 Point 对象，用于指定矩形。

矩形对象同样支持 draw()、undraw()、move()、setFill()、setOutline()、clone() 等方法。此外，矩形还支持的方法包括 r.getP1() 、 r.getP2() 和 r.getCenter()，分别用于获取左上角、右下角和中心，返回值都是 Point 对象。

多边形

多边形类型为 Polygon，创建多边形对象的语句模式为：

```
>>> poly = Polygon(&lt;顶点 1&gt;,..., &lt;顶点 n&gt;)
```

将各顶点用直线相连，即成多边形。

矩形对象同样支持 draw()、undraw()、move()、setFill()、setOutline()、 clone()等方法。此外还支持方法 poly.getPoints()，用于获取多边形的各个顶点。

文本

文本类型为 Text，创建文本对象的语句模式为：

```
>>> t = Text(&lt;中心点&gt;,&lt;字符串&gt;)
```

其中，中心点是个 Point 对象，字符串是显示的文本内容。

文本对象支持 draw()、undraw()、move()、setFill()、setOutline()、clone()等方法，其中 setFill()和 setOutline()方法都是设置文本的颜色。文本对象还支持方法 t.setText(<新字符串>)用于改变文本内容，方法 t.getText()用于获取文本内容，方法 t.setTextColor()用于设置文本颜色。

5.4.3 graphics 与面向对象

在 Tkinter 中，只为画布提供了类 Canvas，而画布上绘制的各种图形并没有对应的类。 因此画布是对象，而画布上的图形并不是对象，至少不是按面向对象风格构造的。graphics 模块就是为了改进这一点而设计的，它将 Tkinter 的绘图功能进行了全面的面向对象包装。 在 graphics 模块中，GraphWin、Point、Circle、Oval、Line、Text 和 Rectangle 等都是类，可以创建相应的对象。每个对象都是相应的类的实例，例如每个具体的“点”都 是 Point 的实例。所有点对象都具有自己的坐标值(x,y)，都支持 getX()、getY()和 draw()等方法（操作）。 为创建一个类的新实例，需要构造器（constructor）。调用构造器的语法模式如下：

```
<类名>(<参数 1>,<参数 2>, ...)
<变量名> = <类名>(<参数 1>,<参数 2>, ...)
```

其中类名指定要创建什么样的实例，例如 Point 或 Circle；诸参数是对象初始化所需的信息，例如 Point 需要两个坐标作为参数，Circle 需要一个点（圆心）和一个数值（半径）作为参数。构造器创建对象后，通常需要将这个对象赋予某个变量，以便今后通过这个变量引用并操作对象。 我们来看一个例子：

```
p = Point(50,60)
```

Point 构造器创建了一个点对象，变量 p 指向这个新创建的点对象。构造器的两个参数表 示点对象的 x 和 y 坐标，这两个值将存储在对象内部的实例变量（instance variable）中（图 5.24）。

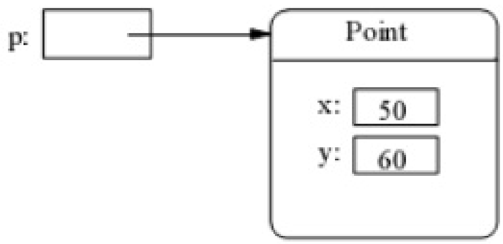


图 5.24 Point 对象的创建 为了请求对象执行其内部定义的方法，需要向对象发消息。例如，对于点对象可以发送消息 p.getX()、p.getY()、p.move(dx,dy)等等。消息的一般形式如下：

```
<对象>.<方法名>(<方法参数 1>,<方法参数 2>, ...)
```

有些对象的实例变量和方法的参数本身也可能是对象。例如，考虑如下语句：

```
>>> win = GraphWin()
>>> c = Circle(Point(100,100), 30)
>>> c.draw(win)
```

上述语句的第一行创建 GraphWin 对象 win。第二行创建 Circle 对象 c，它的圆心是点 对象 Point(100,100），半径为 30。注意，Circle 构造器的第一个参数利用 Point 构 造器创建了圆心点对象。第三行请求 Circle 对象 c 执行它的 draw()方法。图 5.25 显示 了 GraphWin、Circle 和 Point 对象之间的相互关系。我们通常无需关心这些细节，而 只需要创建对象并调用对象的方法，对象自会完成任务，这就是面向对象编程的力量。

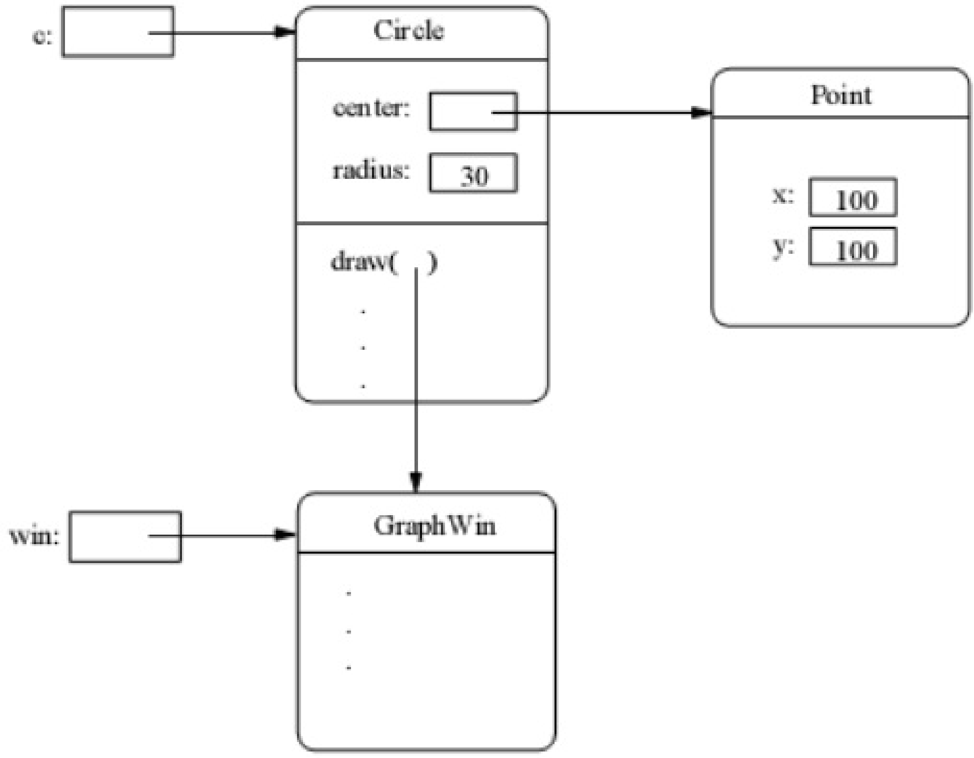


图 5.25 各种对象之间的关系

最后，我们用一个实例演示基于 graphics 模块的图形编程，读者可以自行比较它和Tkinter 编程在风格上的异同。

【程序 5.4】sunmove.py

```
from graphics import * from time import sleep
def main():
    w = GraphWin("Demo",300,200)
    m1 = Polygon(Point(150,199),Point(200,100),Point(250,199))
    m1.setFill('green')
    m1.draw(w)
    m2 = Polygon(Point(200,199),Point(250,80),Point(350,199))
    m2.setFill('green')
    m2.draw(w)
    center = Point(0,100)
    sun = Circle(center,10)
    sun.setFill('red')
    sun.draw(w)
    for i in range(31):
        if i&lt;15:
            sun.move(10,-5)
            center.move(10,-5)
        elif i&lt;20:
            sun.move(10,0)
            center.move(10,0)
        else:
            sun.move(10,5)
            center.move(10,5)
            if i == 30:
                w.setBackground('black')
        sleep(0.25)
    w.getMouse()
    w.close()
main()
```

本程序先创建图形窗口，再画两个多边形和一个圆形（表示两座山和太阳）。然后让圆 形不断移动：先向右上移动，再向右平移，最后向右下移动，显然这是太阳东升西落的模拟。 天黑后点击一下窗口即可关闭窗口结束程序。执行结果如图 5.26 所示。

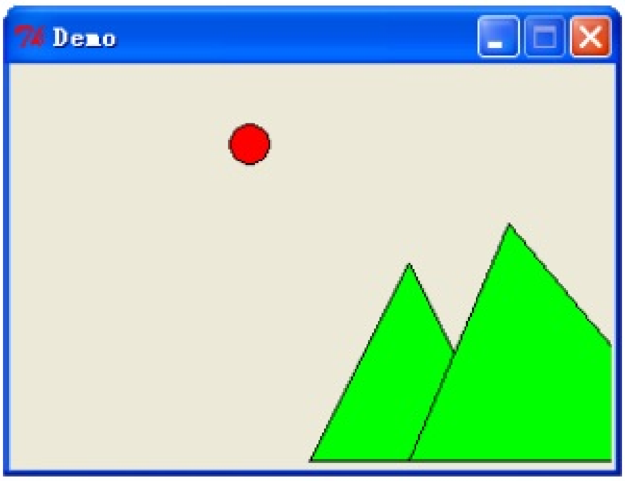


图 5.26 程序 5.4 执行结果截图

5.5 练习

1. 在你的专业中，计算机图形编程可能有什么应用？

2. 为什么说图形是复杂数据？

3. 什么是对象？从你的专业中选择一个研究对象，用程序设计的对象概念来描述它，即列 出它的数据（属性）和操作（方法）。

4. Tkinter 与 graphics 模块的关系是怎样的？

5. 试试在画布上创建汉字文本。如果有乱码，请用汉字的 unicode 编码。

6. 程序设计：画一个射箭运动所用的箭靶。从小到大分别为黄、红、蓝、黑、白色的同心圆，每个环的宽度都等于黄色圆形的半径。

7. 程序设计：绘制奥运五环旗。

8. 程序设计：输入 r 和 y，以 r 为半径画一个圆，以 y 为截距画一条水平直线，然后计算直线与圆的交点。

9. 程序设计：绘制某个数学函数的曲线，例如正弦、余弦、指数函数等等。

10. 程序设计：输入本金和利率，计算 10 年内每一年的本金加利息之和，并用柱状图显示。

11. 程序设计：画一幅冬季景色，有雪人和圣诞树之类。

12. 程序设计：将一圆周进行 n（例如 12 或 15）等分，然后用直线将所有等分点两两连接。

第 6 章 大量数据的表示和处理

第 2 章讨论了现实世界信息在计算机中的抽象表示问题，那里主要介绍的是简单数据， 而本章将继续介绍复杂数据的表示和处理。简单数据一般指单个数据，并且没有内部结构， 不可分割。复杂数据正相反，可在两方面呈现复杂性：一是数量多，即待处理的数据是由大 量相互关联的成员数据组成的；二是有内部结构，即数据在内部由若干分量组成，每个分量 本身可能又由更小的分量组成。对于大量数据，可以用集合体数据类型来表示；对于数据的 内部结构，可以利用面向对象中的类来刻划。本章介绍大量数据的表示和处理，第 7 章介绍 利用类来描述具有深层结构的数据。

6.1 概述

实际应用中所处理的数据经常是“大量同类型数据的集合”，例如一次物理实验获得的 大量实验数据、一篇文章中的所有单词、一幅画布上的所有图形等等，这几个例子分别展示 了大量数值的集合、大量字符串的集合和大量对象的集合。为了表示和处理大量数据，编程 语言提供了集合体数据类型，如 Python 中的列表（list）、元组（tuple）、字典（dict）、集合（set）和文件（file）等。 一个问题中的大量数据通常是“相关的”，即数据之间存在特定的逻辑联系。表示相关的大量数据时，必须将它们之间的逻辑联系也表示出来。在程序设计中，为了方便、高效地 处理大量相关数据，常将各数据按照某种合适的存储结构组织在一起，以便反映各数据之间 的逻辑联系。数据结构（data structure）是计算机科学的一个分支，专门研究如何将大量相 关数据按特定的逻辑结构组织起来，以及如何高效地处理这些数据。

下面以字符串数据为例来说明上面这段话的含义。夸张一点说，字符串数据——如"HELLO"——也是复杂数据，因为它是由一些更简单的数据项（5 个字符"H"、"E"、"L"、"L"和"O"）组成的。为了在程序中能够方便、高效地处理字符串"HELLO"，我们将这 5 个字符视为以“连续存储的序列结构”组织在一起，因为这种存储结构最恰当地反映了 5 个字符之间的逻辑关系，从而最有利于数据处理的实现。反之，如果将 5 个字符东一个西一个地分散存储，比如用几个独立的变量来分别存储这 5 个字符：

```
c1 = "H"
c2 = "E"
c3 = "L"
c4 = "L"
c5 = "O"
```

那么就没有表示出这 5 个字符的逻辑关系（即按特定次序组成的一个单词）。尽管这种存储方式也实现了数据的存储，但它很难支持方便、高效的数据处理。例如，为了输出"HELLO"， 程序必须分别找到 c1 到 c5 这几个存储位置，并取出其中字符组合成输出，显然非常麻烦。 而在“连续存储的序列结构”方式下，程序只需找到一个存储位置即可。

存储数据的目的是为了将来处理数据，故存储结构一定要适应将来的数据处理。对字符 串数据而言，将相关的一组字符存储为一个连续序列就是合适的，因为这种序列结构非常适 合取字符、取子串、合并等字符串操作。因此，编程语言通常都以“连续存储序列”的逻辑 结构来组织多个字符组成的数据，并将字符串作为基本类型提供给程序员使用。这样，程序 员就不必再去费心考虑该用什么逻辑结构来组织多字符数据了。当然，字符串实在算不得复 杂数据，说它是简单数据也无不可，但以上讨论完全适用于真正复杂的数据。

总之，我们得到的教训是：如果不将组成复杂数据的大量数据项按照合适的逻辑关系组 织起来，那么就无法对复杂数据进行方便、高效的处理。换句话说，合适的数据结构往往是设计有效算法的关键。初学编程者通常以为算法才是程序设计的关键，其实不然。计算机科 学家 N. Wirth①提出过一个公式：算法 + 数据结构 ＝ 程序 这足以说明数据结构的重要性。

由于现实世界中构成复杂数据的逻辑关系多种多样，编程语言不可能对每一种逻辑结构 都像对字符串那样提供现成的数据类型和处理方法。另外，即使是同一个复杂数据，随着处 理需求的不同，也会导致采用不同的逻辑结构。因此，编程语言一般不会提供现成的数据结 构给我们使用，我们需要根据待解决的实际问题，自行设计复杂数据的数据结构。

本章介绍用于表示和处理大量数据的集合体数据类型和几种常用的数据结构。

6.2 有序的数据集合体

大量数据按次序排列而形成的集合体称为序列（sequence）。注意，这里所说的“次序”是指各成员数据之间有位置的前后，并非指成员数据按值的大小来排列。就像一群人站成一 排即成序列，并不一定要按高矮顺序排列。

Python 中的字符串、列表和元组数据类型都是序列，第 2 章中对它们有过初步介绍， 本节将继续介绍对序列的处理。从第 5 章我们初步了解了对象的概念，以及如何对图形对象 进行操作。这里要说明的是，Python 序列其实都是以面向对象方式实现的，因此对序列的 处理可以通过对序列对象的方法进行调用而实现。

表 6.1 列出了对序列的一些通用的操作（运算符和内建函数），利用这些操作可以实现 对序列的索引、联接、复制、检测成员等。

方法	含义
s1 + s2	序列 s1 和 s2 联接成一个序列
s * n 或 n * s	序列 s 复制 n 次，即 n 个 s 联接
s[i]	序列 s 中索引为 i 的成员
s[i:j]	序列 s 中索引从 i 到 j 的子序列
s[i:j:k]	序列 s 中索引从 i 到 j 间隔为 k 的子序列
len(s)	序列 s 的长度
min(s)	序列 s 中的最小数据项
max(s)	序列 s 中的最大数据项
x in s	检测 x 是否在序列 s 中，返回 True 或 False
x not in s	检测 x 是否不在序列 s 中，返回 True 或 False

表 6.1 对序列的基本操作

此外，序列还支持比较运算。序列 s 和 t 的大小按字典序确定：首先通过比较 s[0]与 t[0] 来决定大小，相等时再比较 s[1]和 t[1]，依次类推。这就是说，两个序列相等当且仅当它们 的对应位置上的成员都相等，并且长度相同。

各种序列还有各自特殊的操作，下面分别讨论。

> ① PASCAL 语言的设计者。

6.2.1 字符串

关于字符串数据，第 2 章已经详细介绍过对字符串的基本操作，以及利用字符串库 string 提供的函数来实现更丰富的操作。这里我们再介绍另一种处理方式，即面向对象的方式。Python 中，每个字符串实际上都是一个对象，因而可以通过向字符串对象发送方法请求的方式来实现对字符串的操作。表 6.2 列出了字符串对象的一些常用方法，并将对应的 string 库函数（参见表 2.5）列在一起以供比较。

字符串对象方法	**string 库函数**	含义
s.capitalize()	capitalize(s)	s 首字母大写
s.center(width)	center(s,width)	s 扩展到给定宽度且 s 居中
s.count(sub)	count(s,sub)	sub 在 s 中出现的次数
s.find(sub)	find(s,sub)	sub 在 s 中首次出现的位置
s.ljust(width)	ljust(s,width)	s 扩展到给定宽度且 s 居左
s.lower()	lower(s)	将 s 的所有字母改成小写
s.lstrip()	lstrip(s)	将 s 的所有前导空格删去
s.replace(old,new)	replace(s,old,new)	将 s 中所有 old 替换成 new
s.rfind(sub)	rfind(s,sub)	sub 在 s 中最后一次出现的位置
s.rjust(width)	rjust(s,width)	s 扩展到给定宽度且 s 居右
s,rstrip()	rstrip(s)	将 s 的所有尾部空格删去
s.split()	split(s)	将 s 拆分成子串的列表
s.upper()	upper(s)	将 s 的所有字母改成大写

表 6.2 字符串对象的方法

不要忘了，字符串数据是不可修改的，因此表 6.2 中没有修改字符串 s 的方法。 下面通过一些例子来演示字符串对象的方法的使用。

```
>>> s = "I think, therefore I am."
>>> s.count('I')
2
>>> s.find('re')
12
>>> (s.lower()).replace('i','I')
'I thInk, therefore I am.'
>>> s.split()
['I', 'think,', 'therefore', 'I', 'am.']
>>> s.islower()
False
```

6.2.2 列表

我们先回顾第 2 章中介绍的关于列表的知识。 列表是由多个数据组成的序列，可以通过索引（位置序号）来访问列表中的数据。与很

多编程语言提供的数组（array）类型不同，Python 列表具有两个特点：第一，列表成员可以由任意类型的数据构成，不要求各成员具有相同类型；第二，列表长度是不定的，随时可以增加和删除成员。另外，与 Python 字符串类型不同，Python 列表是可以修改的，修改方式包括向列表添加成员、从列表删除成员以及对列表的某个成员进行修改。

作为序列的一种，我们可以对列表施加序列的基本操作，如索引、合并和复制等（参见 表 6.1）。另外由于列表是可以修改的，Python 还为列表提供了修改操作，见表 6.3。

修改方式	含义
a[i] = x	将列表 a 中索引为 i 的成员改为 x
a[i:j] = b	将列表 a 中索引从 i 到 j（不含）的片段改为列表 b
del a[i]	将列表 a 中索引为 i 的成员删除
del a[i:j]	将列表 a 中索引从 i 到 j（不含）的片段删除

表 6.3 列表的修改

本节要引入的是面向对象方式的列表操作。和字符串一样，Python 列表实际上也是对 象，提供了很多有用的方法。例如，append()方法用于向列表尾部添加成员数据：

```
>>> a = ['hi']
>>> a.append('there')
>>> a
['hi', 'there']
```

利用 append()方法，我们可以将用户输入的一批数据存储到一个列表中：

```
data = []
x = raw_input('Enter a number: ')
while x != "":
    data.append(eval(x))
    x = raw_input("Enter a number: ")
```

这段代码实际上是累积算法，其中的列表 data 就是累积器：首先初始化为空列表，然后通 过循环来逐步累积（添加成员数据）。

表 6.4 列出了列表对象的常用方法。

方法	含义
<列表>.append(x)	将 x 添加到<列表>的尾部
<列表>.sort()	对<列表>排序（使用缺省比较函数 cmp）
<列表>.sort(mycmp)	对<列表>排序（使用自定义比较函数 mycmp）
<列表>.reverse()	将<列表>次序颠倒
<列表>.index(x)	返回 x 在<列表>中第一次出现处的索引
<列表>.insert(i,x)	在<列表>中索引 i 处插入成员 x
<列表>.count(x)	返回<列表>中 x 的出现次数
<列表>.remove(x)	删除<列表>中 x 的第一次出现
<列表>.pop()	删除<列表>中最后一个成员并返回该成员
<列表>.pop(i)	删除<列表>中第 i 个成员并返回该成员

表 6.4 列表对象的方法

下面通过例子来说明对列表对象的处理：

```
>>> a = ['Irrational',[3.14,2.718],'pi and e']
>>> a.sort()
>>> a
[[3.14, 2.718], 'Irrational', 'pi and e']
>>> a[0].reverse()
>>> a
[[2.718, 3.14], 'Irrational', 'pi and e']
>>> a.insert(2,'number')
>>> a
[[2.718, 3.14], 'Irrational', 'number', 'pi and e']
>>> print a.pop(0)
[2.718, 3.14]
>>> a
['Irrational', 'number', 'pi and e']
```

编程案例：一个统计程序 对大量数据进行统计、分析是实际应用中常见的问题，通过计算一些统计指标可以获得有关这批数据的多侧面的特征。常用的统计指标包括总和、算术平均值、中位数、众数、标 准差和方差等，这些指标的计算过程具有不同的特性。

“总和”是可以累积计算的，即可以先计算部分数据的和 sum，当有了新数据再加入 sum 并形成新的 sum。重复上述步骤直至所有数据都已加入 sum，这时所得即总和。利用我 们介绍过的累积算法模式，很容易实现求总和的代码：

```
sum = 0
data = raw_input("输入新数据: ")
while data != "":
    x = eval(data)
    sum = sum + x
```

从以上代码可以看到，虽然用户输入了很多数据，但程序中却始终只用一个简单变量data 来存储输入的数据。为什么不怕后面输入的数据将前面输入的数据覆盖掉呢？巧妙之处 在于，累积算法每次接收一个输入数据就立即使用该数据（将新数据加到累加变量 sum 中）， 从而使变量 data 可以用于存储下一个输入数据。我们没有采用“先将所有输入数据存储起 来，然后再求和”的处理策略，因为这个策略需要大量存储空间，更麻烦的是我们预先并不 知道需要多少存储空间。类似地，输入数据的“个数”也可以利用累积算法来求得。

再看“算术平均值”指标，虽然它本身不能直接累积计算，但根据公式“平均值＝总和÷数据个数”可见，可以通过累积算法求得“总和”和“数据个数”，然后直接算出平均值。 推而广之，如果某个统计指标可以表示成某些累积类型指标的代数式，那么这个指标就可以 利用累积算法进行计算，无需保存所有输入数据。

再看一个统计指标——中位数（median）。中位数将全体数据划分为小于和大于它的两 部分，并且两部分的数据个数相等。如果全体数据从小到大有序排列，则处于中间位置的那 个数据就是中位数①。例如，数据集合{3,4,22,50,64}的中位数是 22。中位数的计算与总和、 算术平均值都不同，因为它不能通过累积来计算，如{3,4}的中位数与{3,4,22}的中位数直至{3,4,22,50,64}的中位数基本没什么关系。因此，为了对用户输入的一组数据求中位数，必须将每个数据先保存起来，等全体数据都到位后才能计算。与中位数类似的、不具有累积计算性质的统计指标还有众数、标准差等，可以称之为“整体型”指标，即它们都需要针对全体 数据进行计算。那么，如何存储所有输入数据呢？显然，定义许多独立变量来存储输入数据 是不合适的，因为我们不知道用户会输入多少个数据；即使知道用户将输入 n 个数据，定义 n 个独立变量来存储这些数据也是非常笨拙的做法。其实问题很容易解决，列表可以将所有 输入数据组合成单个数据，这样既保存了所有数据，又不需要定义许多独立变量。

下面我们来编写一个统计程序，其功能是获得用户输入的数值数据，并求出这批数据的 总和、算术平均值和中位数。如前所述，这三个指标分别代表三种类型的统计指标，因此我 们的统计程序虽然简单，但具有一般的意义。

按照模块化设计思想，我们分别为数据输入及每个指标的计算设计一个函数。 首先设计获得输入数据的函数。由于整体型指标中位数的计算需要用到全体输入数据，因此我们先将所有输入数据存储到一个列表中。获得用户输入的关键代码是一个哨兵循环， 数据列表是一个累积器，在循环中逐个接收数据。代码如下：

> ① 若数据个数为偶数，则取处于中间位置的两个数据的平均值。

```
def getInput(): data = []
x = raw_input("Enter a number (&lt;Enter&gt; to quit): ")
while x != "":
    data.append(eval(x))
    x = raw_input("Enter a number (&lt;Enter&gt; to quit): ")
return data
```

接着设计三个统计指标的函数。这些函数的参数都是列表 aList，调用时将存储输入数 据的 data 作为实参传递给 aList 即可。总和及算术平均值很容易计算，只要先对输入列表利 用累积求得总和，然后再除以列表长度即得平均值。列表长度可以用 len()直接求得，不需 要另外

写一个累积循环。代码如下：

```
def sum(aList):
    s = 0.0
    for x in aList:
        s = s + x
    return s
def mean(aList):
    return sum(aList) / len(aList)
```

中位数的计算没有代数公式可用，我们先将全体数据从小到大排序，然后取中间位置的 数据值。当数据个数为奇数时，有唯一的中间位置，故中位数很容易找到；当数据个数为偶 数时，中位数是处于中间位置的两个数据的平均值。列表数据的排序可以利用现成的列表对 象方法 sort()实现，而奇偶性可以利用余数运算的结果来判断。代码如下：

```
def median(aList):
    aList.sort()
    size = len(aList)
    mid = size / 2
    if size % 2 == 1:
        m = aList[mid]
    else:
        m = (aList[mid] + aList[mid-1]) / 2.0
    return m
```

利用以上四个模块，再加上主控模块 main()，就完成了我们的统计程序。完整代码见程 序 6.1。

【程序 6.1】statistics.py

```
def getInputs(): d = []
    x = raw_input("Enter a number (&lt;Enter&gt; to quit): ")
    while x != "":
        d.append(eval(x))
        x = raw_input("Enter a number (&lt;Enter&gt; to quit): ")
    return d
def sum(aList): s = 0.0
    for x in aList:
        s = s + x
    return s
def mean(aList):
    return sum(aList) / len(aList)
def median(aList): aList.sort()
    size = len(aList)
    mid = size / 2
    if size % 2 == 1:
        m = aList[mid]
    else:
        m = (aList[mid] + aList[mid-1]) / 2.0
    return m
def main():
    print "This program computes sum, mean and median."
    data = getInputs()
    sigma = sum(data)
    xbar = mean(data)
    med = median(data)
    print "Sum:", sigma
    print "Average:", xbar
    print "Median:", med
main()
```

6.2.3 元组

第 2 章中简单介绍了元组数据类型，我们知道元组是用一对圆括号括起、用逗号分隔的多个数据项的集合体。元组也是序列的一种，可以利用表 6.1 中的序列操作对元组进行处理。 元组和列表在很多方面都是相似的，但它们有一个重要的不同点：元组不可修改，即不能对元组施加表 6.3 中的操作。如果序列的内容一经创建就不再改变，那么建议使用元组来 表示这个序列，好处是效率较高，而且可以防止出现误修改操作。

元组的括号有时可以省略，例如用在赋值语句中。我们熟悉的为多个变量同时赋值其实 是元组赋值。下面是一些例子：

```
>>> 1,2,3
(1, 2, 3)
>>> x = 1,2,3
>>> x
(1, 2, 3)
>>> x,y,z = 1,2,3
>>> x
1
>>> y,z
(2, 3)
```

元组也可以嵌套，即元组的成员本身可以是元组，例如：

```
>>> t = ("Lucy",("Math",90))
>>> t[1][1]
90
```

Python 是以面向对象的方式实现元组类型的，元组对象支持的方法见表 6.5。

方法	含义
<元组>.index(x)	返回 x 在<元组>中首次出现处的索引
<元组>.count(x)	返回<元组>中 x 的出现次数

表 6.5 元组对象的方法

元组类型的名字 tuple 可以用作构造器，将一个字符串或列表转换成元组对象。例如：

```
>>> tuple('hello')
('h', 'e', 'l', 'l', 'o')
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple(['hello','world'])
('hello', 'world')
```

6.3 无序的数据集合体

如 6.2 节所介绍的，Python 中的列表和元组都是有序集合体，成员之间存在某种次序， 因此可以通过各成员所处的位置（索引）来访问成员，就像一群人站成一排然后报数，每人 报出的数字就是他的位置序号。然而，我们在中学数学里所学的集合（set）是若干元素的 无序集合，集合中各元素之间不存在先后关系，就像一群人散乱地站在一起，无法令其报数。 现实中有很多信息可以用这种无序的数据集合体来表示，Python 提供了两种无序集合体类 型：集合和字典。

6.3.1 集合

Python 提供了集合类型 set，用于表示大量数据的无序集合体。集合可以由各种数据组 成，数据之间没有次序，并且互不相同。可见，Python 集合基本上就是数学中所说的集合①。

集合类型的值有两种创建方式：一种是用一对花括号将多个用逗号分隔的数据括起来； 另一种是调用函数 set()，此函数可以将字符串、列表、元组等类型的数据转换成集合类型的 数据。不管用哪种方式创建集合值，在 Python 内部都是以 set([...])的形式表示的。注意，空 集只能用 set()来创建，而不能用字面值{}表示，因为 Python 将{}用于表示空字典（见 6.3.2 节）。

下面的会话过程演示了集合类型的值的创建。注意，集合中是不能有相同元素的，因此 Python 在创建集合值的时候会自动删除掉重复的数据。

> ① 当然 Python 集合并不完全等同于数学中的集合，例如数学中的集合可能是无穷集。

```
>>> {1,2,3}
set([1, 2, 3])
>>> s = {1,1,2,2,2,3,3}
>>> s
set([1, 2, 3])
>>> set('set')
set(['s', 'e', 't'])
>>> set('sets')
set(['s', 'e', 't'])
>>> set([1,1,1,2,1])
set([1, 2])
>>> set((1,2,1,1,2,3,4))
set([1, 2, 3, 4])
>>> set()
set([])
>>> type(set())
<type 'set'>
>>> type({})
<type 'dict'>
```

集合类型支持多种运算，学过中学数学的读者很容易理解这些运算的含义。我们将常用 的集合运算列在表 6.6 中。

运算	含义
x in <集合>	检测 x 是否属于<集合>，返回 True 或 False
s1 | s2	并集
s1 & s2	交集
s1 – s2	差集
s1 ^ s2	对称差
s1 <= s2	检测 s1 是否 s2 的子集
s1 < s2	检测 s1 是否 s2 的真子集
s1 >= s2	检测 s1 是否 s2 的超集
s1 > s2	检测 s1 是否 s2 的真超集
s1 |= s2	将 s2 的元素并入 s1 中
len(s)	s 中的元素个数

图 6.6 集合运算

下面是集合运算的例子：

```
>>> s1 = {1,2,3,4,5}
>>> s2 = {2,4,6,8}
>>> 6 in s1
False
>>> 6 in s2
True
>>> s1 | s2
set([1, 2, 3, 4, 5, 6, 8])
>>> s1 & s2
set([2, 4])
>>> s1 - s2
set([1, 3, 5])
>>> s1 |= s2
>>> s1
set([1, 2, 3, 4, 5, 6, 8])
>>> len(s2)
4
```

和序列一样，集合与 for 循环语句结合使用，可实现对集合中每个元素的遍历。例如，接着上面的例子继续执行语句：

```
>>> for x in s2:
        print x,
8 2 4 6
```

Python 集合是可修改的数据类型，例如上面例子中修改了集合 s1 的值。但是，Python 集合中的元素必须是不可修改的！因此，集合的元素不能是列表、字典等，只能是数值、字符串、元组之类。同样，集合的元素不能是集合，因为集合是可修改的。然而，Python 另外提

供了 frozenset()函数，可用来创建不可修改的集合，这种集合可以作为另一个集合的元 素。下面的语句展示了 set 和 frozenset 的区别：

```
>>> a = set(['hi','there'])
>>> b = set([a,3])
Traceback (most recent call last):
File "&lt;pyshell#74&gt;", line 1, in &lt;module&gt; b = set([a,3])
TypeError: unhashable type: 'set'
>>> a = frozenset(['hi','there'])
>>> b = set([a,3])
>>> b
set([3, frozenset(['there', 'hi'])])
```

Python 以面向对象方式实现集合类型，集合对象的方法如表 6.7 所示。

方法	含义
s1.union(s2)	即 s1 | s2
s1.intersection(s2)	即 s1 & s2
s1.difference(s2)	即 s1 – s2
s1.symmetric_difference(s2)	即 s1 ^ s2
s1.issubset(s2)	即 s1 <= s2
s1.issuperset(s2)	即 s1 >= s2
s1.update(s2)	s1 |= s2
s.add(x)	向 s 中增加元素 x
s.remove(x)	从 s 中删除元素 x（无 x 则出错）
s.discard(x)	从 s 中删除元素 x（无 x 也不出错）
s.pop()	从 s 中删除并返回任一元素
s.clear()	从 s 中删除所有元素
s.copy()	复制 s

表 6.7 集合对象的方法

接着前面的例子，下面通过集合对象方法的调用来处理集合数据：

```
>>> s2.union([1,2,3])
set([1, 2, 3, 4, 6, 8])
>>> s2.intersection((1,2,3,4))
set([2, 4])
>>> set([2,4]).issubset(s2)
True
>>> s2.issuperset(set([2,4]))
True
>>> s2.add(10)
>>> s2
set([8, 2, 4, 10, 6])
>>> print s2.pop()
8
>>> s2
set([2, 4, 10, 6])
```

6.3.2 字典

在一个数据集合中查找信息有很多种方式，前面介绍的序列采用的是通过位置索引来查 找信息的方式。还有一种常用的查找方式是通过数据间的关联来查找信息，例如手机里的通 信录一般都是通过姓名查找对应的电话号码。Python 中的字典类型可用来实现这种通过数 据查找关联数据的功能。

相信读者都用过字典，知道字典是由大量“词条”组成的，每个词条由“单字（词）” 加“释义”组成。字典的用法正是“根据单字（词）查找释义”。Python 提供的字典类型（dict） 与现实中的字典是类似的：Python 字典是由大量的“键值对（key-value pair）”组成的集合， 每一个键值对形如“key:value”，其用法是通过“键”key 来访问相应的“值”value。

字典类型 dict 与集合类型 set 一样属于无序集合体，即字典中的键值对没有特定的排列 顺序，因此不能像序列那样通过位置索引来查找成员数据。

创建字典

字典的字面值是用一对花括号括起的、以逗号分隔的一些键值对，形如：

```
{k1:v1,k2:v2,...,kn:vn}
```

其中，键值对的“键”可以是任何不可修改类型的数据，如数值、字符串和元组等；而键值 对的“值”则可以是任何类型的数据。不含任何键值对的字典是空字典，表示为{}。例如：

```
>>> d = {'Lucy':1234,'Tom':5678,'Mary':1357}
>>> print d
{'Mary': 1357, 'Lucy': 1234, 'Tom': 5678}
```

注意，字典中键值对的显示次序与定义次序不同，这是因为字典是无序集合，字典的显示次序由字典在内部的存储结构决定。

除了字面值之外，还可以利用类型构造器 dict()来创建字典，创建时需要将字典的键值 对信息作为参数传递给 dict()。参数的形式有两种：一种是关键字参数形式（参见 4.2.4）， 一种是序列（列表或元组）形式，例如：

```
>>> d1 = dict(name="Lucy",age=8,hobby=("bk","gm"))
>>> d1
{'hobby': ('bk', 'gm'), 'age': 8, 'name': 'Lucy'}
>>> d2 = dict([[(5,1),'Worker'],[(6,1),'Child'],[(7,1),'CPC']])
>>> d2
{(5, 1): 'Worker', (6, 1): 'Child', (7, 1): 'CPC'}
```

对字典的操作

字典的主要用途是查找与特定键相关联的值，具体操作形式如下：

```
<字典>[<键>]
```

其返回值就是字典中与给定的键相关联的值。如果指定的键在字典中不存在，则报错（KeyError）。例如：

```
>>> d1["name"]
'Lucy'
>>> d1["age"] 8
>>> d1["hobby"]
('bk', 'gm')
>>> d1["gender"]
Traceback (most recent call last):
File "&lt;pyshell#22&gt;", line 1, in &lt;module&gt; d1["gender"]
KeyError: 'gender'
```

前面创建的字典 d2 是以元组为键的，访问时当然要提供一个元组，且元组括号可省略。 例如：

```
>>> d2[(6,1)]
'Child'
>>> d2[7,1]
'CPC'
```

字典类型的数据是可以修改的。与某个键相关联的值可以通过赋值语句来修改，形如：

```
<字典>[<键>] = <新值>
```

如果指定的键不存在，则相当于向字典中添加新的键值对。例如：

```
>>> d1["age"] = 9
>>> d1
{'hobby': ('bk', 'gm'), 'age': 9, 'name': 'Lucy'}
>>> d1["gender"] = "F"
>>> d1
{'hobby': ('bk', 'gm'), 'age': 9, 'name': 'Lucy', 'gender': 'F'}
```

事实上，创建字典的常用方式就是从空字典开始，利用循环语句以某种方式逐个获得键 值对数据，并利用赋值语句加入字典。

del 命令可以用来删除字典条目，形如：

```
del <字典>[<键>]
```

Python 将字典实现为对象，表 6.8 给出了字典对象的方法。

方法	含义
<字典>.has_key(<键>)	若<字典>包含<键>，返回 True；否则返回 False
<字典>.keys()	返回所有键构成的列表
<字典>.values()	返回所有值构成的列表
<字典>.items()	返回所有(key,value)元组构成的列表
<字典>.clear()	删除<字典>的所有条目

图 6.8 字典对象的方法

下面的会话过程演示了对象方法的用法：

```
>>> d1.keys()
['hobby', 'age', 'name', 'gender']
>>> d1.values()
[('bk', 'gm'), 9, 'Lucy', 'F']
>>> d1.items()[0:2]
[('hobby', ('bk', 'gm')), ('age', 9)]
>>> d1.has_key("gender")
True
```

6.4 文件

众所周知，CPU 只能读写内存，因此当程序运行时，程序所处理的数据必须存储在内 存中。当程序结束或关机、掉电时，内存中的数据就会消失。为了永久保存数据，必须将数 据存储在磁盘、光盘、闪存盘等不依赖于电源的外部存储器上。另外，与外部存储器相比， 内存的容量小而价格高，不适合海量数据存储。总之，计算机问题求解必须考虑如何处理外 部存储器上的大量数据的问题。前面几节介绍的列表、元组、字典等类型虽然可以用于表示 大量数据，但它们都属于内存数据类型，是对内存数据的组织方式。编程语言另外提供了文 件类型来支持大量数据的存储和处理。

6.4.1 文件的基本概念

外部存储器上的数据是以文件形式进行组织的。一组相关数据存储在一起便构成一个文件（file），每个文件被赋予一个文件名，程序通过文件名来访问文件。文件名通常由主名 和扩展名构成，后者用来描述文件内容，如常见的.txt、.jpg、.doc 等等。当外存上存储了大 量文件时，为便于管理，常将文件分组，构成一个个文件夹（或称目录）；如果每个文件夹 中的文件还是很多，则可以继续分组构成子文件夹（子目录），最终形成一个树形层次式目 录结构。

目录路径

为了指定唯一的文件，必须提供详细的路径。事实上，一个完整的文件标识由磁盘驱动 器、目录层次和文件名三部分构成。各部分之间用特定字符进行分隔，分隔字符在不同操作系统中可能是不同的，例如 Windows 使用“\”，而 Unix、Linux 使用“/”。在 Python 程序 中，路径分隔字符既可以使用“\”，也可以使用“/”。例如，Python 安装目录中有个文件 misc.py，其路径可以用字符串

```
"C:\Python27\Lib\compiler\misc.py"
```

来表示。

注意：我们在第 2 章讨论字符串数据时说过，反斜杠字符“\”在 Python 中可作为转义 符，用于表示特殊字符，如"\n"（换行字符）、"\t"（Tab 字符）和"\xc4"（编码为十六进制 c4 的字符）等。文件路径中如果在反斜杠后出现了 n、t、x 等字符，就可能被解释成特殊字符， 从而导致错误。例如，试图用语句

```
>>> f = open("C:\Python27\Lib\compiler\transformer.py")
```

打开文件 transformer.py 时，Python 会将字符串中的\t 解释为 Tab 字符，从而报错。避免这种错误的简单方法是使用正斜杠字符“/”或者使用两个反斜杠“\”表示单个反斜杠，即形 如

```
"C:/Python27/Lib/compiler/transformer.py" "C:\\Python27\\Lib\\compiler\\transformer.py"
```

如果文件和程序在同一个文件夹中，则程序中可以省略文件路径，直接使用文件名来标识文件。

文件格式

文件中存储的数据可以有不同的格式。最简单的文件是文本文件，其中存储的数据是无 格式的字符串，因此对文本文件的处理可以逐字符（字节）地进行。另一种文件格式是二进 制文件，其中存储的数据是二进制串，这种二进制串当然不能按一个字节对应一个字符的方 式来

解释，例如存储图像、音频信息的.jpg、.mp3 文件就是常见的二进制文件。至于.doc、.xls 和.ppt 等格式的文件各自具有独特的文件结构，也可以归入二进制文件类别，只能用专门的程序来处理。

在信息管理应用中，大量信息的组织通常都采用“字段－记录－文件”的层次格式。字 段是最基本的不可分割的数据项，如学号、姓名、年龄等；记录是若干个相关字段结合在一 起形成的数据，例如将某个学生的基本信息组合起来就构成形如（5120309001，张三，18） 的记录；大量同类型的记录即构成了文件，例如全体学生记录存储在磁盘上即构成一个学生 数据文件。所有记录按顺序存储，则文件格式可用图 6.1 表示。



图 6.1 字段－记录－文件

本书只讨论文本文件的处理。文本文件中存储的字符主要是可打印字符，包括字母、数字、标点符号和空格等。但有一些控制字符也是常用的，例如“回车”、“换行”等字符，可 用来将文本内容组织成一行一行的形式。由于控制字符不是可打印字符，在程序中只好用转 义符来来间接地表示，例如回车符表示为"\r"，换行符表示为"\n"。

6.4.2 文件操作

常用计算机的人都知道，许多应用软件（如 Word、媒体播放器等）都需要处理文件， 并且都需要经过打开文件、读写文件、关闭文件的步骤，这其实是程序设计中文件处理的一 般过程的反映。

打开文件

在读写文件之前首先需要“打开”文件，这个步骤可以简单地理解为对磁盘文件进行必 要的初始化，至于其底层细节则无需了解。

Python 提供了函数 open 用于文件打开，用法如下：

```
f = open(<文件名>,<打开方式>)
```

其含义是按指定的<打开方式>打开由<文件名>标识的磁盘文件，创建一个文件对象作为函 数的返回值，并使变量 f 引用这个文件对象。常用的打开方式包括"r"和"w"，它们分别表示“读”方式和“写”方式。

顺便强调一下，Python 中的文件处理是面向对象风格的，即文件是一个对象，通过文 件对象的方法来实现文件操作。我们在第 5 章中初步介绍了对象概念，并且将在第 7 章详细 讨论面向对象。

为了读取一个文件的内容，需要以读方式打开文件。例如：

```
f = open("oldfile.dat","r")
```

成功执行后，就可以通过文件对象 f 来读取文件 oldfile.dat 的内容了。若指定的文件不存在，则 Python 将报错（IOError）。

为了向一个文件中写入内容，需要以写方式打开文件。例如：

```
f = open("newfile.txt","w")
```

成功执行后，就可以通过文件对象 f 来向文件 oldfile.dat 中写入内容了。注意，以写方式打开文件时，如果指定的文件不存在，则创建该文件；如果指定的文件已经存在，则会清除该文件原来的内容，即相当于创建新文件。所以，以写方式打开文件时一定要小心，不要把现有文件破坏了。

读文件

在介绍文件读写之前，先要理解文件“当前读写位置”的概念。读者应该了解老式的录 放机的录放过程吧：录放机有一个磁头，用于读取或录入磁带信息，随着磁带的转动，磁头 也就不断改变着录放位置。Python 中的文件采用类似的顺序读写过程：打开文件后，当前 读写位置就是文件开始处；随着读写命令的执行，当前读写位置不断改变，直至到达文件末 尾。

Python 中的文件对象提供了 read()、readline()和 readlines()方法用于读取文件内容。 read()的用法如下：

```
<变量> = <文件对象>.read()
```

含义是读取从当前位置直到文件末尾的内容，并作为字符串返回。如果是刚打开的文件对象， 则返回的字符串包含文件的所有内容。

read()方法也可以带有参数：

```
<变量> = <文件对象>.read(n)
```

含义是读取从当前位置开始的 n 个字符，并以此字符串作为返回值。如果指定的 n 大于文件中从当前位置到末尾的字符数，则仅返回这些字符。如果当前位置已到达文件末尾，则 read 返回空串。

假设有一个文件 rhyme.txt，其文本内容是：

```
Good, better, best, Never let it rest, Till good is better, And better, best.
```

下面的语句序列对此文件进行读取

```
>>> f = open("rhyme.txt","r")
>>> s = f.read(8)
>>> s
'Good, be'
>>> f.read(20)
'tter, best,\nNever le'
>>> print f.read()
t it rest,
Till good is better, And better, best.
>>> f.close()
```

readline()的用法如下：

```
<变量> = <文件对象>.readline()
```

含义是读取从当前位置到行末（即下一个换行字符）的所有字符，并以此字符串作为返回值， 赋值给变量。通常用此方法来读取文件的当前行。如果当前处于文件末尾，则 readline 返回 空串。例如：

```
>>> f = open("rhyme.txt","r")
>>> s = f.readline()
>>> s
'Good, better, best,\n'
>>> f.readline()
'Never let it rest,\n'
>>> print f.readline()
Till good is better,
>>> f.close()
```

readlines()的用法如下：

```
<变量> = <文件对象>.readlines()
```

其含义是读取从当前位置直到文件末尾的所有行，并将这些行构成一个字符串列表作为返回值，列表中的每个元素都是文件的一行。如果当前处于文件末尾，则 readlines 返回空列表。例如：

```
>>> f = open("rhyme.txt","r")
>>> f.readline()
'Good, better, best,\n'
>>> f.readline()
'Never let it rest,\n'
>>> f.readlines()
['Till good is better,\n', 'And better, best.\n']
>>> f.readlines()
[]
```

写文件

当文件以写方式打开时，可以向文件中写入文本内容。与读文件一样，写入位置也是由 “当前读写位置”决定的。Python 文件对象提供两种写文件的方法：

```
<文件对象>.write(<字符串>)
<文件对象>.writelines(<字符串列表>)
```

其中，write 的含义是在文件当前位置处写入字符串，writelines 的含义是在文件当前位置处依次写入列表中的所有字符串。 下面的语句序列创建了一个新文件，并向其中写入了李白的名诗：

```
>>> f = open("d:/libai.txt","w")
>>> f.write("窗前明月光")
>>> f.write("疑是地上霜\n")
>>> f.write("举头望明月\n 低头思故乡")
>>> f.close()
```

注意每一次 f.write()都是紧接着上次写入的内容继续的，并不会因为是另一条 f.write()就另 起一行。为了写多行文本，必须人工添加换行字符“\n”。那么，上述语句序列所创建的文 件 libai.txt 有几行文本呢？没错，只有 3 行，因为第一次调用 f.write 时并没有写入换行符， 这导致诗的前两句被写在同一行上了。如图 6.2 所示。

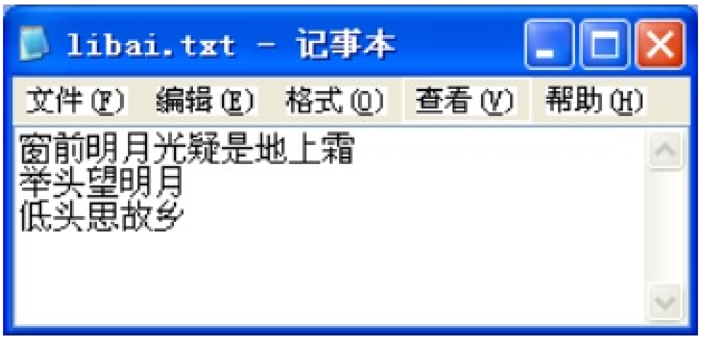


图 6.2 写入多行文本

再次强调，写方式打开文件会导致要么创建一个新文件，要么清除一个旧文件，总之文件的内容是全新的。那么有没有办法在现有文件内容基础上再写入一些新内容呢？答案是肯 定的。Python 还提供一种文件打开方式"a"，表示“追加”。以追加方式打开文件后，当前位 置被定位在文件末尾，可以继续写入文本而不改变原有的文件内容。例如：

```
>>> f = open("d:/libai.txt","a")
>>> f.write("\n---- 李白《静夜思》")
>>> f.close()
```

结果如图 6.3 所示。

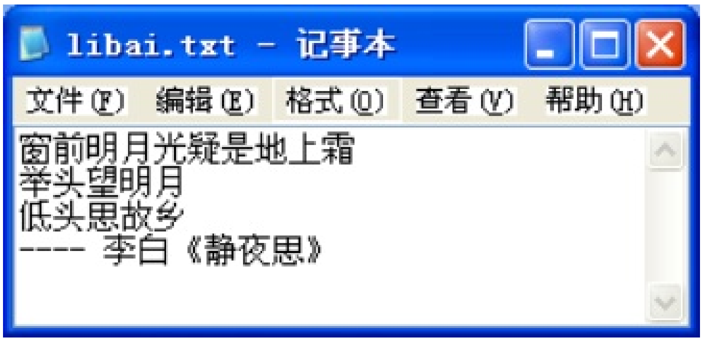


图 6.3 向文件追加写入内容

关闭文件

文件处理结束后需要关闭文件，这个步骤大体上涉及释放分配给文件的系统资源，以便 分配给其他文件使用。通过调用文件对象的 close 方法来关闭文件：

```
<文件对象>.close()
```

注意，即使程序中没有关闭文件，Python 程序结束时也会自动关闭所有打开的文件。

然而好的做法是由程序自己关闭文件，否则有可能因程序意外终止而导致文件数据丢失。例如，以写方式打开文件时，如果向文件中写入了文本但还没有关闭文件，那么所写内容是不会存盘的。这时再以读方式打开同一文件，read()命令返回的是空串。下面的语句序列演示了这种情况。

```
>>> f = open("d:/test","w")
>>> f.write("some words")
>>> g = open("d:/test","r")
>>> g.read()
''
>>> f.close()
>>> g.seek(0)
>>> g.read()
'some words'
```

所以，强烈建议读者在程序中一旦结束对文件的读写，就立即关闭文件。

文件处理程序的常见结构

许多应用程序的算法结构都属于直接了当的 IPO（输入－处理－输出）模式，当输入输 出都是文件时，程序的结构大体如下：

```
infile = open("input.dat","r")
outfile = open("output.dat","w")
while True:
    text = infile.readline()
    if text == "":
        break
    do something with text ...
    outfile.write(data)
infile.close()
outfile.close()
```

此代码的核心是一个 while 循环，循环的每一步利用 readline()读取输入文件的一行，然后对该行进行处理，并将处理结果写入输出文件。当某次循环读到空行（视为文件尾），则利用 break 跳出循环体，从而结束对文件的处理。

除了“while 循环+readline()”的结构，还可以利用“for 循环+readlines()”的结构。readlines() 一次性读出所有行，形成一个列表，然后针对这个列表进行循环。

```
for line in infile.readlines():
    do something with line
    ...
```

实际上，Python 语言甚至允许直接将打开的文件与 for 循环结合使用，达到和“for 循 环+readlines()”同样的效果。代码如下：

```
infile = open("input.dat","r")
for line in infile:
    do something with line
    ...
```

这种用法有个好处是无需考虑内存大小，而 readlines()要求内存足够大，以便容纳它返回的列表。

向文件追加数据

前述读方式打开的文件只能读取不能写入，写方式打开的文件是新建文件（写打开现存文件的话将清除内容），只能写入不能读取。有没有办法保留现存文件的内容并加入新内容 呢？

一种做法是先将文件的现有数据利用 readlines()读出来存入一个列表，然后向该列表添 加数据，最后再把新列表写入文件。这种做法对小文件没有问题，但当文件大小为数百 MB 或若干 GB 时，为了保存所有行的列表需要消耗大量内存。

其实 Python 还提供了一种打开方式"a"，称为“追加”方式，可以用于在现存文件的尾 部追加新数据。当然，如果请求打开的文件不存在，"a"方式就和"w"方式一样，创建一个新 文件。下面的语句演示了追加方式的用法：

```
>>> f = open("oldfile.txt","a")
>>> f.write("something new\n")
>>> f.close()
```

6.4.3 编程案例：文本文件分析

本节讨论一个文件分析程序，其功能是输入一个文本文件，对文件内容进行分词（将字符流划分为单词），然后统计文件中的字符数、单词数、每个单词的出现次数以及行数，最 后输出统计结果。按出现频率前 n 名的单词。这种分析在很多应用中都会用到，例如自然语 言处理、文档相似性比较、搜索引擎等。

分析程序的算法设计是直接了当的，其核心是对多个指标进行累积计数。其中，对字符 数和行数的计数可以利用文件操作的结果直接得到：read()可将整个文件的内容作为一个字 符串返回，字符串长度就是字符总数；readlines()将文件的所有行构成一个列表返回，列表 长度就是行数。至于单词总数，需要先将文件内容（字符串）划分成单词，这可以利用 string 库中的 split 函数实现。既可以对 read()返回的整个字符串分词，也可以通过循环来对 readlines() 返回的每一行字符串分词，我们将采用更简单的前一种方法。下面是实现这一部分工作的示 意代码，其中 f 表示被分析的文件对象：

```
numchars = len(f.read())
numlines = len(f.readlines())
numwords = len(string.split(f.read()))
```

分析程序中最麻烦的是对每个单词出现次数的累积计数。按照过去介绍的累积算法模式，需要为每一个累积量定义一个累积变量，并在循环中不断更新该变量。然而，这种做法 并不适合现在的场合，因为为文件中可能出现的成千上万个单词各定义一个累积变量显然太 笨拙了，更何况文件中到底有哪些单词是不能预知的。编程解决问题的诀窍之一是使用合适 的数据类型，6.1.2 中介绍的字典正可以在这个场合派上用场。

我们将建立一个字典 worddict，其关键字是文件中出现的单词，值是该单词在文件中出 现的次数，即 worddict[w]等于 w 在文件中出现的次数。在读文件单词的过程中，每当遇到 单词 w，就用下面的语句递增 w 的计数值：

```
worddict[w] = worddict[w] + 1
```

不过这里还有一个小麻烦：当首次遇到单词 w 时，字典 worddict 中尚未建立相应的词条， 即 worddict[w]无定义，因此上述递增计数的语句将导致错误（KeyError）。为解决这个小麻烦，最容易想到的是用条件语句来检测单词 w 是否已经存在于字典中，代码如下：

```
if worddict.has_key(w):
    worddict[w] = worddict[w] + 1
else:
    worddict[w] = 1
```

另一种做法是利用例外处理，通过捕获关键字错误（KeyError）来决定是递增计数还是 首次建立词条。代码如下：

```
try:
    worddict[w] = worddict[w] + 1
except KeyError:
    worddict[w] = 1
```

这个做法在使用字典的程序中很常用，我们的分析程序也采用了这个做法。 除了核心代码，还需补充一些在分词之前对文件字符串进行预处理的代码。其一，将文件内容中的字母都转换成小写，以使单词"WORD"和"word"被识别为同一单词；其二，将文 件内容中的各种标点符号都替换成空格，以使单词"one,two"能被正确地划分为两个单词 "one"和"two"，以及"one, two"不被划分为"one,"和"two"①。做这两件事的代码如下：

```
text = string.lower(text)
for ch in "`~!@#$%^&*()-_=+[{]}\\|;:'\",&lt;.&gt;/?":
    text = string.replace(text,ch," ")
```

接下来即可划分单词，并对所有单词进行循环，在循环过程中构造字典 worddict。代码如下：

```
wordlist = string.split(text)
worddict = {}
for w in wordlist:
    try:
        worddict[w] = worddict[w] + 1
    except KeyError:
        worddict[w] = 1
```

最后输出分析结果。由于单词可能很多，我们的分析程序只示意性地输出了 5 个单词及 其出现次数。更好的做法是根据出现次数对单词排名，并输出最频繁的前 n 名单词，有兴趣 的读者可以试着完善这个功能。

将以上讨论综合起来，即得完整的文件分析程序。

> ① 这里的细微差别在于逗号后是否有空格。

【程序 6.2】textanalysis.py

```
import string
def main():
    fname = raw_input("File to analyze: ") f = open(fname,"r")
    text = f.read() numchars = len(text) f.seek(0)
    numlines = len(f.readlines()) text = string.lower(text)
    for ch in "`~!@#$%^&*()-_=+[{]}\\|;:'\",&lt;.&gt;/?":
        text = string.replace(text,ch," ")
    wordlist = string.split(text) numwords = len(wordlist) worddict = {}
    for w in wordlist:
        try:
            worddict[w] = worddict[w] + 1
        except KeyError:
            worddict[w] = 1
    print "Number of characters:",numchars print "Number of lines:",numlines print "Numbe
    for i in range(10):
        print pairlist[i],
main()
```

注意，由于需要两次读文件（read 和 readlines），所以在第二次读文件之前应将“读写头” 移动到文件开始处，这就是第 8 行的 f.seek(0)所做的事情。

假设有文件 yours.txt，其内容如下：

```
The life that I have Is all that I have,
And the life that I have Is yours.
The love that I have Of the life that I have
Is yours, and yours, and yours.
A sleep I shall have, A rest I shall have,
Yet death will be but a pause. For the peace of my years
In the long green grass,
Will be yours, and yours, and yours.
```

则运行程序 6.2 后，将得到如下结果：

```
File to analyze: yours.txt Number of characters: 315
Number of lines: 14 Number of words: 70
('and', 5) ('all', 1) ('peace', 1) ('love', 1) ('is', 3)
```

6.4.4 缓冲

当一个人饿了，面对一大碗饭，他该怎么吃呢？任务的目标是将这一碗饭送到肚子里去， 解决饿的问题，而达成目标的最快方法是将一碗饭一口吞下，可惜没人有这么大的嘴。事实上，人们采取的是每次吃一口的方式，一口一口地将饭吃到肚子里去。这个例子很好地说明了计算机解决问题时的“缓冲”技术。

利用计算机解决问题时，经常需要将大量数据从一个地方传送到另一个地方，并且一次 性地传送所有数据会遇到种种限制。这时，可以在内存中建立一个缓冲区（buffer），用做传 送数据的临时过渡。通过缓冲区，就可以将大量数据以一小批一小批的方式传送到目的地。

例如，处理一个很大的磁盘文件时，由于内存容量有限，无法一次性将文件内容全部读 入内存，只好在内存中建立一个缓冲区，每次将一小批数据读入缓冲区以供 CPU 处理。上 面说的吃饭例子中，我们的嘴就是缓冲区。生活中类似的例子很多，例如学生用的书包其实 也是一个缓冲区——学生不可能随身带着自己的所有书籍，于是采用书包作为临时存储区， 每天只需带当天要用的课本。

又如，当计算机向打印机传送数据进行打印时，由于发送方（计算机）和接收方（打印 机）的数据处理速率存在很大差异，不可能将数据一下子传给打印机，这时也可以使用缓冲 区来协调计算机和打印机的步调。这种情形在生活中也很常见，去银行办理业务时，由于顾 客来的频繁，而银行职员处理业务较慢，不可能实现“随到随处理”，因此银行设置了等待 区，用于缓冲顾客流。

下面我们编写一个文件拷贝程序，功能是将用户指定的文件复制到文件夹 d:\backup 中。 假设内存容量有限或者 CPU 处理能力有限，导致每次只能处理 1024 个字符。为此，我们使 用 read(n)来读文件，其中参数 n 表示从文件读取 n 个字符。程序代码如下：

【程序 6.3】buffer.py

```
def main():
    fname = raw_input("Enter file name: ")
    f = open(fname,"r")
    fcopy = open("d:/backup/"+fname,"w")
    while True:
        buffer = f.read(1024)
        if buffer == "":
            break
        fcopy.write(buffer)
    f.close()
    fcopy.close()
```

显然这里的字符串变量 buffer 相当于缓冲区，通过每次读入 1024 个字符，像蚂蚁搬家一样将整个文件复制到目的文件夹。

6.4.5 二进制文件与随机存取*

前面介绍的文件处理是针对文本文件的，并且主要是顺序存取文件。本节简单介绍二进 制文件的处理以及文件的随机存取。

二进制文件

任何文件在底层都是字节序列。文本文件的字节可解释成字符的编码：如果是 ASCII 编码，则每个字节表示一个字符；如果是 GBK 编码，则每两个字节表示一个汉字。对文本 文件的处理完全基于这种字符解释。而二进制文件的字节序列表示任意的二进制数据，不能 解释为字符序列。对二进制文件的处理也必须基于特定的解释来进行。

Python 语言支持对二进制文件的处理，处理过程仍然是“打开－读写－关闭”三部曲。 打开二进制文件时必须指明“以二进制方式打开”，具体就是用"rb"、"wb"和"ab"分别表

示读打开、写打开和追加打开。例如：

```
>>> bf1 = open("c:/windows/notepad.exe","rb")
>>> bf1.read(10)
'MZ\x90\x00\x03\x00\x00\x00\x04\x00'
>>> bf2 = open("c:/windows/explorer.exe","rb")
>>> bf2.read(10)
'MZ\x90\x00\x03\x00\x00\x00\x04\x00'
```

这里我们分别打开了两个常用的 Windows 应用程序文件：记事本和资源管理器，并且各读 了头 10 个字节的内容。从输出结果可见，这些字节一般不能解释成字符①。细心的读者还可 以发现，notepad.exe 和 explorer.exe 这两个文件的头 10 个字符是一样的。这一点都不奇怪，因为它们都是 exe 文件，而 exe 文件是有规定的文件头格式的。作为练习，读者不妨以二进制方式打开几个.jpg 文件，并读取文件头若干字节的数据，看看有什么发现。

当然我们还可以将二进制文件以"wb"和"ab"方式打开，从而可以修改二进制文件。不过 除非你知道自己在做什么，一般不要尝试修改二进制文件，因为可能破坏文件格式。

关闭二进制文件和关闭文本文件是一样的，调用文件对象的 close 方法即可。

文件的随机存取

文件一般都是顺序读写的，即从文件开始处按顺序读写文件内容直至文件尾。然而，有时候也需要对文件进行随机读写，即直接定位到文件的特定位置进行读写，不需要读写从文 件头到目标位置之间的内容。以读书作类比，顺序读写就像从第一页逐词逐行读到最后一页 一样，而随机读写则像跳跃式读书，略过中间所有内容直接翻到某一页。

我们说过，读写文件时可以想象有一个“读写头”，就像磁带录音机的磁头一样，当前 读写头所在位置决定了读写的内容是什么。刚打开文件时，读写头位于文件开始处；随着读 写语句的执行，读写头不断移动。顺序读写就像磁带录放机在进行正常的回放或录音，而随 机读写就像快进和快倒。

Python 文件对象提供的 seek()方法可用于文件的随机存取，其用法形如

```
<文件对象>.seek(n)
<文件对象>.seek(n,m)
```

其中，seek(n)的含义是将文件当前位置移到偏移为 n 的地方，这里的偏移是相对于文件开始位置的，即文件的第 1 个字节偏移为 0，第 2 个字节偏移为 1，依此类推。seek(n,m)的含义是将文件当前位置移到偏移要依 m 值来定：m 为 0 时相对于文件开始位置（即与 seek(n)相同），m 为 1 时相对于文件当前位置，m 为 2 时相对于文件末尾。 偏移为正数表示朝文件尾方向移动，偏移为负数表示向文件头方向移动。

> ① 二进制文件中也可以含有字符数据，例如 exe 文件的头两个字节是字母 MZ，这是 exe 文件的标志。

下面的语句序列首先创建一个汉字文本文件 ccfile.txt，其中每个汉字（包括标点符号） 占用 2 字节。其次，以读方式打开 ccfile.txt，然后文件当前位置移到偏移 12 处（即略过前 5 个汉字和 1 个逗号）并读取 4 个字节（即“处处”）；然后倒退 16 个字节并读取 2 个字节（即“春”）；最后向前移动 26 个字节并读 2 个字节（即“风”），最后显示三次读的内容所联接 而成的字符串“处处春风”。

```
>>> f = open("ccfile.txt","w")
>>> f.write("春眠不觉晓，处处闻啼鸟。夜来风雨声，花落知多少。")
>>> f.close()
>>> f = open("ccfile.txt","r")
>>> f.seek(12)
>>> s = f.read(4)
>>> f.seek(-16,1)
>>> s = s + f.read(2)
>>> f.seek(26,1)
>>> s = s + f.read(2)
>>> print s
处处春风
>>> f.tell()
30L
```

顺便说一下，文件对象还提供 tell()方法，用于确定当前读写位置。具体用法见上面演 示的最后两行，显然读完“风”后，读写头即停留在 30 号字节处。

6.5 几种高级数据结构*

以上介绍的各种数据集合体都是 Python 直接提供的数据类型，属于基本的数据结构。本节介绍几种高级数据结构，编程语言不直接支持它们的表示和操作，需要程序员自己实现。

6.5.1 链表

如前所述，列表是由许多数据按次序排列形成的一种数据结构，列表成员之间的逻辑关 系是由他们的排列次序表示的。例如，如果一群人按姓氏笔画坐在一排相邻的椅子上，那么 这些人的排列次序就表示了他们姓氏笔画的关系，排在 1 号座位的人肯定是笔画最少的，排 在 i 号座位上的人肯定比排在 i+1 号座位上的人笔画要少（参见图 6.4）。



图 6.4 用排列次序表示数据间逻辑关系

这种连续排列的数据结构的优点是：仅凭排列次序（或相邻关系）就知道成员数据之间的逻辑关系，而不需要另外存储表示成员间逻辑关系的信息；可以通过位置信息（索引）对 任何成员进行随机访问，而不需要从头开始一个一个查看。但连续存储结构有也有缺点：如 果需要增加新成员，必须移动大量数据以便为新成员腾出空间；如果要删除某个数据， 删除 后必须移动大量数据以便填补空缺、保持连续性。仍以图 6.4 所示场景为例，如果新来了一 个姓“冯”的人要加入队列，按数据逻辑关系他应当坐在“王”“孙”之间，因此必须使“王” 以后的所有人向右移动一个座位；如果“郑”离开了，那么“周”和其后的所有人必须左移 一个座位。可见，插入、删除操作的代价很大。

再看另一种场景：仍然是一群人要按姓氏笔画顺序排列，但这些人是东一个西一个随便 站立着的，因此无法仅凭这些人所处的位置来判断谁笔画多谁笔画少。这种情形下还有没有 办法表示他们的姓氏笔画顺序信息呢？当然有，例如我们可以让每个人用手指着应该排在他 后面的那个人（图 6.5）。这样，虽然这群人站得杂乱无章，但是通过他们的手指，事实上形 成了一个有序的排列。注意，最后一个人没有可指的对象，我们不妨让他以手指地，表示这 是排列的末尾。

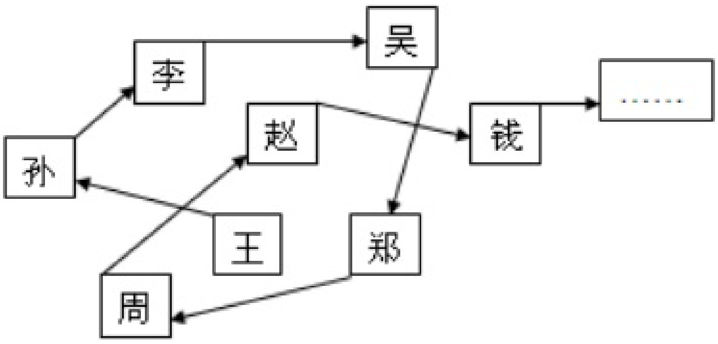


图 6.5 用链接表示数据间逻辑关系

图 6.5 形象地表示了一种以链接方式组织的列表，这种数据结构称为链表（linked list）。 在链表中，成员之间的逻辑关系不是通过存储位置的相邻来表示，而是通过专门的链接信息 来表示。我们将链表中的成员称为结点，每个结点都由两部分信息组成：结点的数据和结点 的链接。结点的数据是实际应用要处理的数据，而结点的链接是对另一个结点的引用（或称 指

针），用于表示数据间的逻辑关系。链表中最后一个成员的链接必须设置为表示“无所指” 的某个特殊值。链表结构的第一个结点是整个链表的入口，通常用一个专门的变量来记录链 表入口。链表的形状如图 6.6 所示。



图 6.6 链表

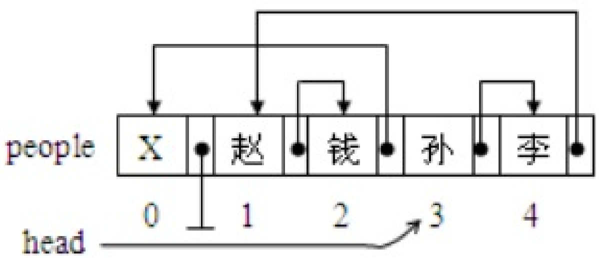
链表可以很好地解决连续存储列表的缺点。例如，如果图 6.5 中新来了“冯”，那我们 只需让“王”的手指改为指向“冯”，并让“冯”指向“孙”；如果“郑”要离开，我们只需 让“吴”的手指改为指向“周”！

然而，与普通列表相比，链表在访问其成员数据时比较麻烦，因为无法通过位置信息来 随机访问链表成员。例如，我们无法直接读取“链表的第 5 个结点”，为了进行这个操作， 必须从链表的头开始，顺着链接向后逐个检查结点。

编程实例：链表的表示和处理

有的编程语言提供了指针类型（存储单元的物理地址），可以很方便地表示链表结点之间的链接。但链接实际上是逻辑层的概念，不必非得用物理层的指针来实现。下面通过前述 按姓氏笔画排序的例子来说明链表的表示及操作方法，其中链接是以结点在列表中的位置索 引实现的。

我们用包含两个成员的列表[(name,strokes),link]来表示结点，其中第一个成员本身是二 元组，分别存储姓氏 name 和笔画数 strokes，第二个成员是链接 link。所有结点存储在列表 people 中，这里 people 相当于动态分配的存储空间，结点在 people 中的位置索引就是结点的“存储地址”，结点的 link 值就是另一个结点的位置索引。因此，虽然结点是按随机次序 存储的，但所有结点按其 link 值前后相连就形成了一个链表。图 6.7(a)展示了存储空间中各 结点的物理存储次序和由链接决定的逻辑次序，其中各个结点的值如图 6.7(b)所示。我们另 外用变量 head 指向链表头（此处即索引为 3 的“孙”结点）。



地址	结点
0	[('X',100),-1]
1	[('赵',9),2]
2	[('钱',10),0]
3	[('孙',6),4]
4	[('李',7),1]

(a) (b)

图 6.7 链表的表示

读者应该注意到，people 中存储的第一个结点很特别，这是我们设计的代表链表尾的特 殊结点。链表尾结点包含一个笔画数高达 100 的假想姓氏 X，目的是使将来新结点总能在链 表尾之前找到插入位置，这样可以使程序代码更简明。

现在来看如何在链表中插入新结点。我们首先利用 people.append()方法在存储空间的尾 部建立新结点 N，然后再将 N 插入到链表中。具体插入过程是：从链表头 head 开始，沿着 链接 link 查看链表，将沿途各结点与 N 比较，直至找到第一个笔画数大于 N 的结点 M。然 后使 N 的 link 指向 M，而原先指向 M 的结点改为指向 N（参见图 6.8）。这样的 M 肯定能 找到，因为最坏情况下会找到链表尾，而那里有一个笔画数为 100 的结点①。

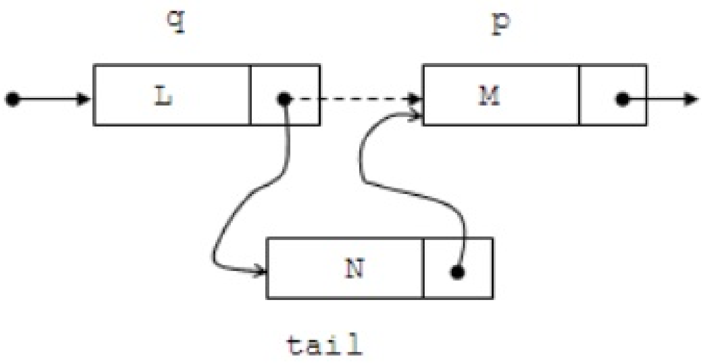


图 6.8 向链表中插入新结点

如图 6.8 所示，为了在结点 L 和结点 M 之间插入结点 N，需要调整 L 和 N 的 link 值， 为此需要在查找链表的过程中记下连续两个结点 L 和 M 的地址，这正是下列代码中变量 p 和 q 的任务。插入结点的主要代码如下：

```
p = head
q = -1
while True:
    if people[p][0][1] <= people[tail][0][1]:
        q = p
        p = people[p][1]
    else:
    people[tail][1] = p
    if q &gt;= 0:
        people[q][1] = tail
    else:
        head = tail
    break
```

> ① 据说笔画数最多的汉字是由四个“龍”组成的，共 64 画。

解决了结点插入链表的问题，则链表的创建问题就变得很平凡了。从空链表（实际上有 一个特殊的链表尾结点）开始，每次根据用户输入的姓氏和笔画数建立新结点，并调用结点 插入算法，重复这个过程即可创建整个链表。程序 6.4 实现了这个功能。

【程序 6.4】linkedlist.py

```
from string import split
def insert(llist,head,tail):
    p = head
    q = -1
    while True:
        if llist[p][0][1] <= llist[tail][0][1]:
            q = p
            p = llist[p][1]
        else:
        llist[tail][1] = p
        if q >= 0:
            llist[q][1] = tail
        else:
            head = tail
        break
    return head
def main():
    people = [[('X',100),-1]]
    head = 0
    s = raw_input("Enter name and strokes: ")
    while s != "":
        s2 = split(s,',')
        name,strokes = s2[0],eval(s2[1])
        people.append([(name,strokes),-1])
        tail = len(people)-1
        head = insert(people,head,tail)
        s = raw_input("Enter name and strokes: ")
    print "Physical order:",
    for i in range(1,len(people)):
        print people[i][0][0],
    print
    print "Logical order:",
    p = head
    while people[p][1] >= 0:
        print people[p][0][0],
        p = people[p][1]
main()
```

主程序首先创建空链表（实际上包含特殊的链表尾结点），然后由用户按“姓氏，笔画”格 式输入数据，程序在 people 末尾建立对应的新结点（相对于为新结点分配存储空间），接着 调用 insert 函数将新结点插入到链表中。重复输入数据、存储新结点、插入新结点的过程直 至输入为空，最后分别按 people 中的结点次序（物理存储次序）和链接的次序（逻辑次序） 显示所有结点的姓氏。下面是程序的一次执行过程和结果：

```
Enter name and strokes: 赵,9
Enter name and strokes: 钱,10
Enter name and strokes: 孙,6
Enter name and strokes: 李,7
Enter name and strokes: 周,8
Enter name and strokes: 吴,7
Enter name and strokes: 郑,8
Enter name and strokes: 王,4
Enter name and strokes:
Physical order: 赵 钱 孙 李 周 吴 郑 王
Logical order: 王 孙 李 吴 周 郑 赵 钱
```

图 6.9 是最终结果的示意图。

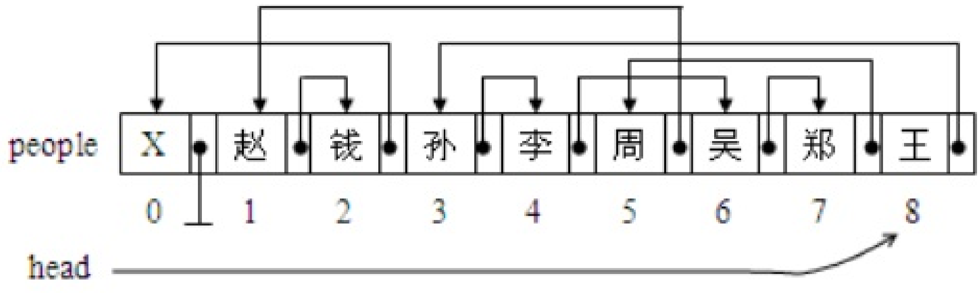


图 6.9 输入 8 个姓氏之后的结果

程序 6.4 只实现了链表的插入功能，作为练习，读者可以尝试为程序增加查找、删除等 功能。

以上介绍的是最简单的单链表。为了更有效地处理链表，还可以设计双链表、循环链表 等结构。事实上，利用链接，还可以设计各种各样的非线性数据结构，如树和图等等。有关 内容可阅读数据结构教材。

6.5.2 堆栈

堆栈（stack）也是一种数据集合体，其中的数据构成一种具有“后进先出（LIFO）”性 质的数据结构，即最后加入堆栈的数据总是首先取出。现实中堆栈的例子俯拾皆是，例如碗橱里的一摞碗、纸箱里的一摞书、弹夹中的子弹等等（图 6.10），他们共同的特点是先放进 去的东西垫底，最后放进去的东西在顶上，而取东西的顺序正好相反。



图 6.10 现实中的堆栈例子

如果忽略各种具体堆栈中无关紧要的成分，如所堆放的东西（碗、书、子弹）、容器（纸箱、碗橱、弹夹）和放入/取出的具体实现（人工、机械），那么我们可以抽象地定义堆栈。所谓堆栈，是以如下两个操作进行处理的数据结构：

- push(x)：在堆栈顶部推入一个新数据 x，x 即成为新的栈顶元素；
- pop()：从堆栈中取出栈顶元素，显然被取出的元素只能是最后加入堆栈的元素。 为了完善这两个操作，还需提供一些辅助操作，如：
- isFull()：检查堆栈是否已满。如果堆栈具有固定大小，那么满了之后是无法执行 push() 的；
- isEmpty()：检查堆栈是否为空。如果堆栈是空的，那么 pop()操作将出错。

此前介绍的数据类型大多是具体的，即它们的实现方式是给定的，例如 int 类型是以 4 个字节来表示，字符串类型是特定编码的字节串等等。而现在我们所讨论的堆栈则是抽象数 据类型，因为我们只规定了堆栈的操作方式，并没有规定操作的具体实现方式。

在具体应用中，可以采用多种不同的方式来实现堆栈这个抽象数据类型。例如，可以采 用列表来实现堆栈。令列表 stack 是存放数据的堆栈，按照堆栈的要求，对 stack 只能执行 push 和 pop 操作，不能像列表那样可以随机存取任何一个元素。假设以列表头为栈底，以 列表尾为栈顶，那么向堆栈中放入元素就只能在尾部添加，Python 列表对象提供的 append 方法正好提供堆栈所需的功能，因此可以用 append 来实现 push()，形如：

```
def push(stack,x):
    stack.append(x)
```

另外，Python 列表对象的 pop()方法的功能是取出列表的最后一个元素，恰好符合堆栈的 pop()方法的要求，因此可以这样实现堆栈 pop 操作：

```
def pop(stack):
    return stack.pop()
```

为了防止从空堆栈中取数据的错误，我们定义一个检测堆栈是否为空的函数：

```
def isEmpty(stack):
    return (stack == [])
```

利用上述以列表实现堆栈的技术，程序 6.5 首先通过用户输入数据创建一个堆栈，然后 再逐个取出堆栈成员。输出恰好是输入的逆序，这是由堆栈的 LIFO 性质决定的。可见，利 用堆栈来逆序显示列表数据是非常容易的。

【程序 6.5】stack.py

```
def push(stack,x):
    stack.append(x)
def pop(stack):
    return stack.pop()
def isEmpty(stack):
    return (stack == [])
def main():
    stack = []
    print "Pushing..."
    x = raw_input("Enter a string: ")
    while x != "":
        push(stack,x)
        x = raw_input("Enter a string: ")
    print "Popping..."
    while not isEmpty(stack):
        print pop(stack),
main()
```

下面是程序 6.5 的一次执行情况：

```
Pushing...
Enter a string: 1st
Enter a string: 2nd
Enter a string: 3rd
Enter a string: 4th
Enter a string: Popping...
4th 3rd 2nd 1st
```

堆栈在计算机科学中非常有用，一个常见的用例是实现表达式的计算。 读者都熟悉算术表达式的中缀形式，但在用计算机处理表达式时常将表达式写成后缀形式，例如“1 + 2”可写成“1 2 +”、“3 + 4 *5”*可写成*“3 4 5* +”。后缀形式的表达式可以 利用堆栈来非常方便地求值，算法如下：

1. 扫描后缀形式的表达式，每次读一个符号（运算数或者运算符）；

2. 如果读到的是运算数，则 push 到堆栈中；如果读到的是运算符，则从堆栈 pop 两个运算数，并执行该运算，然后将运算结果 push 入堆栈；

3. 重复 1、2，直至到达表达式尾。这时堆栈中应该只剩一个运算数，就是表达式的结果值。

图 6.11 显示的是“3 4 5 * +”的计算过程。

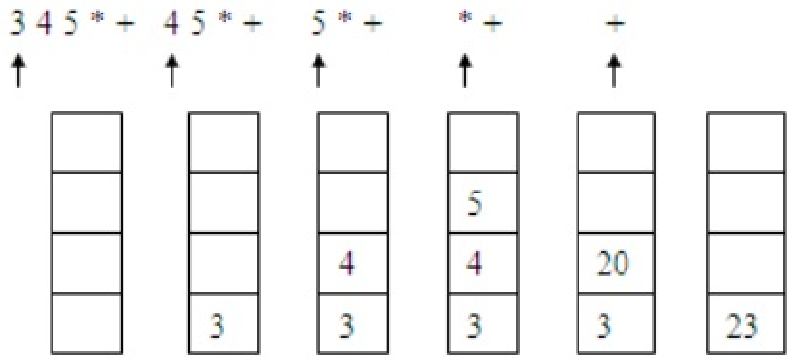


图 6.11 利用堆栈对后缀表达式求值

以上求值部分非常容易实现，但要想对用户输入的中缀形式的算术表达式进行求值，还 需要先对输入进行语法分析，拆分出运算符和运算数，然后改成后缀形式。这部分编程有点 复杂，所以在此我们就不实现这个程序了。有兴趣的读者可以尝试解决这个问题。

6.5.3 队列

队列（queue）也是数据集合体，其中的数据成员有序排列。与堆栈的“后进先出”相 反，队列具有“先进先出（FIFO）”的性质，即最先加入队列的数据将最先移出队列。现实 生活中，当很多人等待某项服务时，通常需要排队，这就是队列，排在最前面的人最先获得 服务。参见图 6.12。

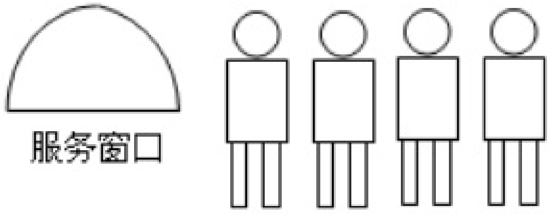


图 6.12 队列

队列也是一种抽象数据类型，完全由操作定义其特性。与堆栈的 push/pop 类似，队列的两个主要操作是：

- enqueue：入队，即在队列尾部添加数据；
- dequeue：出队，即将队列头部的数据移出队列作为返回值。

队列的具体实现有多种方式，例如可以用顺序列表、链表来实现队列。由于队列和堆栈 具有相似性，所以这里不展开介绍了。作为练习，读者可以模仿上一节的例子，用列表来实 现队列。

6.6 练习

1. 分别举例说明现实中的什么信息适合用列表、元组、集合、字典来表示和处理。

2. 以统计指标的计算为例，说明为什么同样是处理大量数据，有的程序不需要使用数据集 合体来存储大量数据，而有的程序则需要。

3. 给定两个列表 `s1 = [2005,7,2,8]` 和 `s2 = [’L’,’u’,’c’,'y']`，计算以下表达式：

（1） `s1 + s2`

（2） `2 * s1`

（3） `s1[1]`

（4） `s2[1:3]`

（5） `s1[1] + s2[-1]`

4. 对第 3 题中的列表进行以下操作后，s1 和 s2 的值是什么。各小题之间是独立的。

（1） `s1.remove(2)`

（2） `s1.sort().reverse()`

（3） `s1.append([s2.index(’c’)])`

（4） `s2.pop(s1.pop(2))`

（5） `s2.insert(s1[2],’I’)`

5. 修改程序 6.1，使程序能计算更多统计指标（如标准差）。

6. 程序设计：自己编程实现 Python 列表对象的方法。

（1）编写函数 `count(aList, x)`，功能同 `aList.count(x)`；

（2）编写函数 `isin(aList, x)`，功能同 `x in aList`；

（3）编写函数 `index(aList, x)`，功能同 `aList.index(x)`；

（4）编写函数 `reverse(aList)`，功能同 `aList.reverse()`。

7. 编写函数 `shuffle(aList)`，其功能是像洗牌一样将列表打乱。

8. 利用筛法找出小于等于 n 的所有质数。基本思想是：首先创建从 2 到 n 的数值列表；然 后将列表的第一个数显示输出（是质数），并从列表中删除该数的所有倍数。重复以上过程 直至列表为空。例如，如果 n 为 10，则初始列表为[2, 3, 4, 5, 6, 7, 8, 9, 10]。输出 2，并删除

2、4、6、8、10。现在列表为[3, 5, 7, 9]。输出 3，删除 3、9。现在列表为[5, 7]。输出 5，并删除 5。最后对[7]输出 7，删除 7。至此列表为空，程序结束。

9. 设计一个学生信息管理系统。每个学生包括学号、姓名、年龄等信息，大量学生数据存 储在文件中。程序开始后向用户显示一个操作菜单，包括向数据文件增加数据项、删除数据项、查找数据项和退出等功能。用户选择菜单项后执行相应功能，执行完毕回到菜单界面。

11. 利用链表来实现堆栈数据结构。

12. 分别利用普通列表和链表来实现队列数据结构。

第 7 章 面向对象思想与编程

面向对象思想和方法具有强大的描述复杂数据和构建复杂系统的能力，因此面向对象编 程已成为当今流行的编程范型，是大多数程序员在解决问题时的不二之选。第 5 章中通过图 形对象初步介绍了对象概念，本章将系统地介绍面向对象思想和面向对象编程。

7.1 数据与操作：两种观点

任何计算机程序都是对特定数据进行特定处理的过程。当我们利用计算机解决问题时，

不外乎要做两件事情：一是将问题要处理的数据表示出来，这可以借助编程语言提供的基本数据类型、复杂类型构造手段以及更高级的逻辑数据结构等来实现；二是设计对这些数据进行处理的算法过程，并利用编程语言提供的各种语句编制成一步一步执行的操作序列。因此，用计算机解决问题的关键是确定问题所涉及的数据以及对数据的操作。

关于数据和操作这两部分的关系，在程序设计思想和方法的发展过程中存在两种不同的观点：一种是传统的以操作为中心的面向过程观点，一种是现代的以数据为中心的面向对象观点。

7.1.1 面向过程观点

我们用一个简单程序来说明传统程序设计的思维方式。

【程序 7.1】eg7_1.py

```
x = 1
y = 2
z = x + y
print z
```

到目前为止，我们在编程时基本上都是这样思考的：先用特定数据类型的常量或变量来 表示数据（如程序 7.1 中分别存入变量 x 和 y 的整数类型值 1 和 2），然后再利用合适的操作（如程序 7.1 中的加法运算“+”）按一定的步骤来处理数据。在这种思考方式下，数据和对 数据的操作被看作是分离的两件事情：数据只是信息的表示，不表达任何操作，在程序中处于“被动”地位；而对数据的操作在程序中则处于“主动”地位，是驱动程序实现特定功能 的力量。程序 7.1 可视为用操作“+”主动地去处理被动的数据 x 和 y，从而实现加法功能。 图 7.1 以一个比喻来形象地展示这种观点：数据与操作之间的关系正如心与箭的关系——没 有丘比特的箭，两颗心是不会彼此连结的。

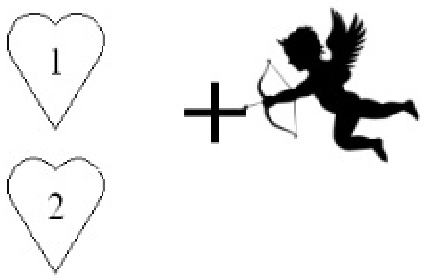


图 7.1 传统观点：数据与操作分离

在数据与操作分离的传统观点下，通常以算法过程的设计为主线来展开程序设计，故可称为以过程为中心的程序设计。以求解一元二次方程的程序 3.6 为例，数据（系数 a、b、c） 明确后，需要精心设计的是处理这些数据的操作过程：先计算判别式 b2-4ac，然后根据判别 式的值判断方程是否有解，有解的情况下再利用公式求解，最后输出结果。

在以操作为中心的设计理念下，程序中的数据有时对整个操作过程都是公开的，可以被 操作过程中的每个步骤访问、处理。例如，假设程序 7.1 的操作不是单一的加法，而是在加法操作（第 3 行）之后还有两个操作：

```
w = x - y
z = z * w
```

可以看出，数据（x、y、z、w）对程序中所有的操作都是公开的。这时，程序中对数据的访问不受任何控制，非常容易出现错误的操作。

当然，实际的应用程序不会像程序 7.1 这样简单。复杂程序中不但数据复杂，而且对数 据的操作也非常复杂，所有操作可能形成漫长而曲折的过程。为了应付操作过程的复杂性， 按照第 4 章所介绍的模块化编程思想，可以将复杂操作过程组织成为若干个较小的模块—— 函数，每个函数实现一个相对简单的、单一的功能。例如下面这个“复杂”程序①：

【程序 7.2】eg7_2.py

```
def op1(a,b):
    return a * a - b * b
def op2(a,b):
    return a ** b + b ** a
x = 1
y = 2
z = 3
result1 = op1(x,y)
result2 = op2(x,z)
print result1 + result2
```

从一个更抽象的层次看，每个函数其实相当于一个操作。与程序 7.1 相比，程序 7.2 对 更多的数据（x、y、z）进行更复杂的操作：先执行 op1 操作，再执行 op2 操作，最后输出结果。无论是程序 7.1 的简单操作“+”还是程序 7.2 的复杂操作“op1”、“op2”，它们都是“对 数据的操作”，仍然属于“数据与操作相互分离”的思考方式，整个程序仍然是对数据按一定 顺序执行操作的过程。

不过，作为高抽象级别操作的函数具有一定的访问控制能力。函数就像一个提供某种功 能的黑箱，使用者需要的只是它的功能，并不需要知晓它内部是如何实现功能的。函数内部 处理的数据不是对整个程序都公开的数据，一个函数不能访问另一个函数内部的数据。然而， 程序中仍然有一些全局数据是对所有操作（包括函数）公开的，仍然存在前述访问控制问题，例如程序 7.2 中的两个函数 op1 和 op2 都在处理数据 x。

总之，不管程序是简单还是复杂，不管操作是语句级的还是函数级的，传统程序设计都 是按照数据与操作分离的观点，以过程为中心展开程序设计的。在这种面向过程的编程范型 中，强调的是对数据的操作过程，程序员思考的主要问题是数据如何表示、对各数据执行什 么操作以及各操作的先后顺序如何安排。当程序很复杂时，可以采用自顶向下设计和模块化 设计方法，将使用低级别操作的复杂过程设计成使用高级别操作的简单过程。

要指出的是，基于数据与操作分离观点的面向过程编程具有其内在的局限性，在处理某 些复杂问题和系统时显得不合适。例如，图形用户界面（GUI）程序②就不属于“对给定数据，

> ① 这个程序自然一点也不复杂，但不妨碍它可以用于说明复杂操作的问题。
>
> ② 详见第 8 章。

按特定次序对数据进行操作，操作完毕程序即告结束”的程序执行模式。以 Word 程序为例，当启动 Word 打开文档（即程序数据）之后，接下去对数据如何操作呢？Word 程序并不知道该做什么，它只能等在那里。接下去用户可能用键盘输入文本，也可能用鼠标点击菜单或工具栏进行存盘或打印，总之用户需要通过某种交互事件来告诉 Word 程序该如何操作数据，一个操作完成后 Word 又进入等待状态。可见，GUI 程序属于“先建立一个图形界面，然后 等

待来自用户的不可预知的事件发生；事件发生后才导致执行某些操作，事件处理完毕又回 到等待状态”这样一种程序执行模式，程序从启动到结束的具体执行过程取决于事件发生的 顺序，不像传统程序那样预定义了执行顺序。

为了适应 GUI 程序这类没有明确的预定义操作次序、靠不确定的事件来驱动的程序和系 统的开发，提高开发效率和质量，计算机科学家提出了一种新的观点来看待数据与操作之间 的关系，即面向对象的观点。

7.1.2 面向对象观点

什么是面向对象？要回答这个问题，首先要理解面向对象思想中最基本的观点：数据和对数据的操作不可分离。

其实这个观点对我们来说并不完全陌生。通过第 2 章介绍的数据类型的概念，我们已经 意识到：特定的数据值与能对该数据执行的操作是密切关联的。对于数值型数据，合法的操 作不外乎加减乘除之类；对于字符串数据，合法的操作不外乎查找串中字符或子串、改变字 母大小写之类。脱离数据的类型来考虑操作是没有意义的，例如在 Python 中单独的一个操作符“+”的意义是不确定的，因为数值、字符串和列表类型的数据都能施加意义不同的“+” 操作。即使是对同为数值型的整数和浮点数数据，除法操作“/”也有不同的含义。

除了语言本身提供的基本数据类型，对于复杂数据类型同样可以说明数据与操作的密切 关联。例如，圆形是一种复杂的数据类型，对圆形可以施加求面积、移动位置等操作；而对 于一个由姓名、年龄、考试成绩等多个简单数据项组合而成的学生数据，可以施加查询姓名、计算平均绩点等操作，绝不会提出计算学生面积的要求。

总之，数据与对数据的操作确实是紧密相关、不可分离的。既然如此，那我们干脆将数 据和操作两者结合在一起，抽象出一种实体：该实体拥有一些数据，同时也知道如何对这些 数据进行操作。这种数据和操作结合在一起所形成的实体称为对象（object）。图 7.2 展示了这种思考方式，与图 7.1 相比，现在心、箭合为一体，就好比青年男女不是等待丘比特的撮 合，而是自备弓箭，随心而动。

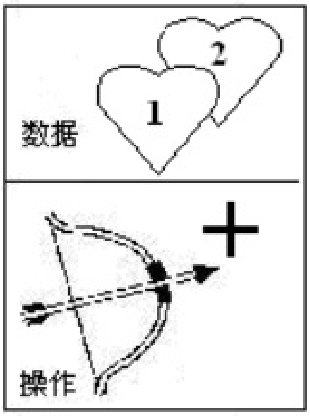
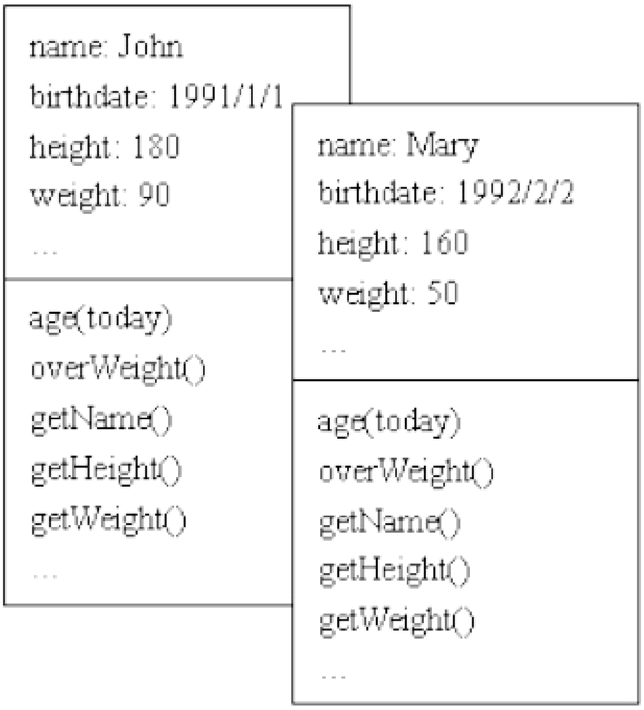


图 7.2 对象是数据和操作的结合体 可以将对象视为广义的“数据”，因为对象里确实存储着数据。但与传统数据不同的是，对象自己掌控对自己存储的数据的处理方法，而不是由外部来决定如何处理。外部如果想对 某个对象存储的数据进行操作，只能向对象发送一个表示操作请求的消息（message），然后由对象来响应这个请求，执行被特定的操作，并将结果告知请求者。显然，对象并不是对什 么消息都能做出响应，对象能够响应的消息由该对象能够执行的操作决定。对象将它能响应 的消息对外公布，就像一个服务机构对外公布服务项目，这些消息（可执行的操作）构成了 对象与外部进行交互的界面（interface，也称接口），外部只能通过这个界面与对象打交道。 基于对象概念来分析问题和设计解法，这就是 面向对象编程 （ object-orientation programming，简称 OOP）。通过 OOP 所得到的程序是一个由很多

对象组成的系统，可以向 对象发送消息来实现对数据的处理，全体对象通过相互协作来完成程序的数据处理功能。而 传统的面向过程编程，得到的程序是一组对数据进行操作的过程，通过按顺序执行这些过程来实现程序功能。 面向对象是强大的分析问题、解决问题的思维工具，因为“对象”这个概念可以用来抽象、描述现实世界的几乎一切事物，例如人、电视机、汽车等等。可以说，世界是由各种对 象组成的，每个对象都具有一些数据特性和一些操作行为，了解了对象的数据特性和操作行 为就认识了对象。作为例子，我们来看“人”为什么可视为“对象”：第一，每个人都具有自 己的数据，如姓名、出生日期、身高、体重等；第二，每个人对他的数据都有自己的操作方 法，例如通过计算当前日期与出生日期的差值来得到年龄、通过公式“标准体重 =（身高－ 100）× 0.9）”来判断自己是否超重等。而且每个人都能响应外部发来的消息（如询问年龄 的消息），也就是执行相应的数据操作。再看一个例子，“电视机”也可视为“对象”：第一， 每台电视机都具有自己的数据，如型号、尺寸、频道数目等；第二，每台电视机都有自己的 数据操作，例如开机、关机、调频道、调音量等。而且电视机能够响应外部发来的消息并执 行相应的操作，如按下遥控器上的某个按键就能让电视机执行执行调频道的操作。图 7.3 中 给出了两个“人”对象和一个“电视机”对象的示意图。



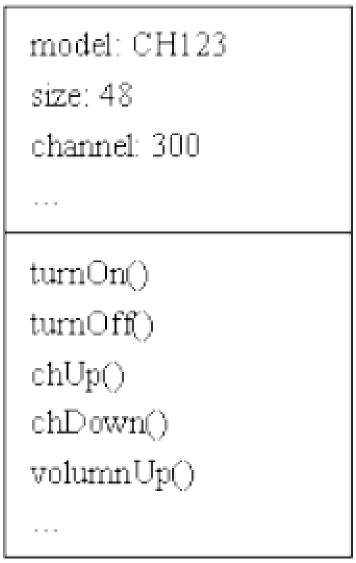


图 7.3 两个“人”对象和一个“电视机”对象

除了“人”、“电视机”这种有形的、具体的对象，我们也可以将无形的、抽象的事物看作是对象。例如可以将“室内环境”视为对象：该对象的数据包括温度、湿度、容积等，该 对象能够响应的操作包括提高温度（具体也许是通过空调设备）、增加湿度（具体也许是通过 人工洒水）、换算容积单位等。

综上所述，我们将数据和对数据的操作融为一体，形成具有静态信息和动态行为的对象。 以面向对象的观点去描述现实世界，就是要将现实世界刻画成由各种对象组成，并且各对象之间进行交互、协作的系统。

7.1.3 类是类型概念的发展

如上所述，对象可以视为广义的数据，因此和普通数据一样属于某种数据类型。从图 7.3 可以看出，“人”和“电视机”就属于两种完全不同的对象类别，而 John 和 Mary 这两个“人” 对象则具有完全相同的数据构成和操作，只是各自的数据值不同。

用计算机解决问题时，首先需要明确问题中涉及哪些数据，并在程序中将这些数据用编 程语言提供的数据类型表示出来，然后再去考虑需要对这些数据执行何种操作。为了表示数 据，编程语言一般提供若干基本数据类型（如 Python 的 int、float、str 和 list 等类型），并为 这些基本类型提供相应的基本操作（如 Python 中对 int、float、str 和 list 都提供了+运算，尽 管含义不同）。

然而，实际问题中往往涉及很复杂的数据，不能用基本数据类型直接表示。为了表示复 杂数据，大体有两种办法：一种是将复杂数据分解成若干个简单数据项，以便每个数据项可 以用基本类型表示；另一种是由用户自定义新的数据类型，以便对复杂数据进行直接的、整 体的表示。例如，如果要表示一个学生的姓名，可以简单地用一个字符串数据表示；如果要 表示一个学生的年龄，可以简单地用一个整数数据表示。但如果要整体表示一个“学生”，包 括该学生的姓名、年龄、地址等信息，就没法用基本数据类型直接表示了。一种解决办法是 将整体的“学生”分解成姓名、年龄、地址等简单数据，并通过分别处理这些简单数据而达 到处理“学生”数据的目的。但这不是好办法，因为这种表示法丢失了数据的整体性，在维 护姓名、年龄、地址等数据间的联系时很麻烦。另一种解决办法是将学生整体视为一个数据 值，并为这种数据值定义新的数据类型（因为编程语言中没有现成的类型能够表示该数据）。

假设我们要为“学生”数据定义一个新的数据类型 S，那么 S 应该是由若干更简单的数 据项构成的（如学号、姓名等），我们称这些构成 S 的成员数据为 S 的属性。除了定义 S 类 型数据的属性，还需要定义能对 S 数据执行什么操作（如修改姓名或年龄、读取地址等）。 可以利用编程语言提供的基本类型和新类型定义机制来实现 S，例如用 str 类型表示姓名和学 号，用 int 类型表示年龄之类，用函数实现对 S 数据的操作。定义了 S，就好像为编程语言添 加了一个新的数据类型，应用程序就可以像使用整数、字符串等基本类型一样去使用 S。

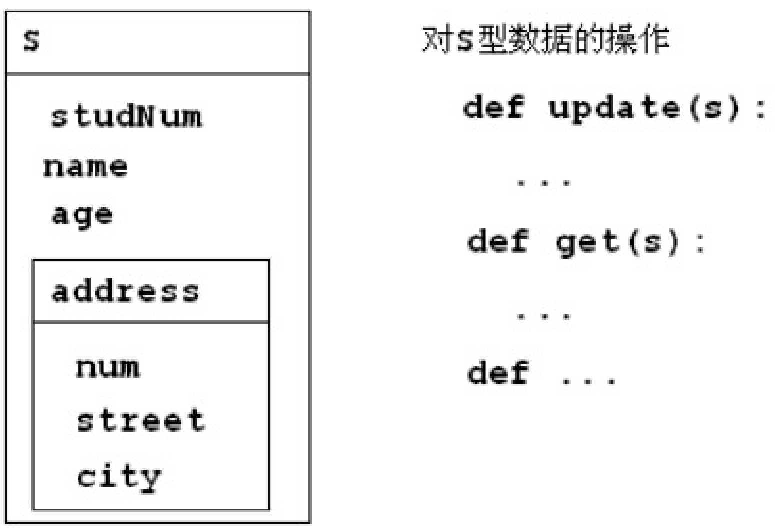


图 7.4 复杂数据类型 S 是多个属性的组合体

在传统观点下，由于数据和操作被视为分离的，因此定义新数据类型时只需定义复杂数据是怎样构成的，例如将“学生”数据定义成学号、姓名、年龄、地址的组合体①。至于如 何操作这种复杂数据，则需要另行编写处理代码，例如写一个函数 update(s)来实现对 S 型数据的修改，写另一个函数 get(s)来实现对 S 型数据的读取等等。总之，数据类型 S 与对这种 类型的数据的操作 update()、get()等是分离的两件事情，设计也是分离的（如图 7.4 所示）。 顺便提一下，将一些数据组合起来构成更复杂的数据的过程是可以重复进行的，即组合体成 员本身可以是复杂数据，如图 7.4 中的属性 address 一样。而在面向对象观点下，数据与操作 是不可分离的，是同一实体（即对象）的两个侧面。因此，当用户为复杂数据定义新的数据 类型 T 时，必须同时描述 T 的值的构成以及对 T 型值的操作。这样，上面的“学生”类型 S 将被定义成如图 7.5 所示的样子：

> ① 这种组合体在不同编程语言中有不同术语，如 Pascal 语言的记录类型和 C 语言的结构类型。



图 7.5 复杂数据类型 S 是多个属性及操作的组合体

由此，我们从传统的数据类型概念发展出了“类”的概念。类（class）是广义的数据类型，能够定义复杂数据的特性，包括静态特性（即数据）和动态特性（即对数据的操作方法）。 正如传统类型 int 可视为由 3、525 等合法整数值组成的集合一样，类也规定了它的合法值的 形式和范围。类的值就是“对象”，也称为类的实例。例如图 7.5 中的类 S 的合法值就是每一 个学生。

早期编程语言在创建新类型方面比较弱，但随着数据类型概念的发展，现代编程语言大 多都提供了强大的自定义数据类型的语言构造。面向对象编程语言中的类定义机制使得自定 义数据类型的能力达到了比较完善的程度。

7.2 面向对象编程

OOP 的特色包括抽象、封装、消息、模块化、多态性、继承等。

7.2.1 类的定义

利用 OOP 来解决问题时，首要任务是确定该问题涉及哪些对象，每种对象分别具有什 么数据和操作。类就是对这些信息的表达。类的创建，体现了面向对象的诸多思想和方法， 本节对此进行详细介绍。

抽象

人们在认识客观世界时，经常采用抽象方法来对客观世界的众多事物进行归纳、分类。 抽象就是忽略事物中与当前目标无关的、非本质的特征，而抽取与当前目标有关的、本质的特征。经过抽象，能够确认事物间的共性，并将具有共性的事物归入一类，从而得到一个抽 象概念。抽象时需要考虑“当前目标”，对于某个事物，根据不同的解题目标，可以进行不同 的抽象。

抽象是面向对象的基本方法，而抽象的工具就是“类”。我们用类来抽象、描述待求解 问题所涉及的事物，具体包括两个方面的抽象：数据抽象和行为抽象。数据抽象描述某类对 象共有的属性或状态，行为抽象描述某类对象共有的行为或功能特征。

例如，对学校里一个个具体的学生张三、李四、王五等进行概括，可以抽取出学号、姓 名等数据属性，还可以抽取出选课、做项目、加入学生社团等行为属性，从而建立“学生类”。 又如，对马路上形形色色的汽车进行概括，可以抽取出制造商、型号、排量等数据属性，还 可以抽取出启动、刹车、加速等行为属性，从而建立“汽车类”。

抽象可以是在多个层次上进行的。例如，当抽象出学生、教师、职工等类别之后，可以 从他们进一步抽象出“师生员工”类；当建立了汽车、火车、飞机等类别之后，可以从他们 进一步抽象出“交通工具”类。如此进行，最终可以形成一个抽象层次，称为类层次。这种 抽象方法在各学科中都是常用的，最典型的如生物学中的分类层次：

```
智人 → 人科 → 灵长目 → 哺乳纲 → 脊索动物门 → 动物界
```

封装

在日常生活中，人们使用着各种提供特定服务的设备，如汽车、彩电等等。对于这些设备，人们通常只需了解它们的功能，而不关心它们的内部是怎样工作的。类似地，计算机程 序也是提供特定服务的“设备”，我们使用程序时通常也只关心程序的功能，而不关心程序内 部的实现过程。OOP 中的封装概念正是这种思想的体现。

封装（encapsulation）是指用类将对象的数据和操作结合成一个封闭的程序单元，并对 外部隐藏内部实现细节。隐藏信息包含两重意思：一是用户无需了解隐藏的信息就能使用该 类，二是不允许用户直接操作类中被隐藏的信息。

封装导致外界不能直接存取对象的数据，但这并不是说我们就无法处理对象数据了。与 数据封装在一起的还有对数据的操作（称为方法的一些函数），类会对外公开这些方法的名称 和调用格式，亦即提供了与外界的交互界面。外界可以向对象发消息（所请求的方法名称及 参数），然后对象通过执行方法来响应外界的消息。所以，外界可通过消息机制来间接处理对象数据。当然，外界只能按对象允许的方式来处理对象数据，因为对象能够响应的消息是由类定义决定的。

封装对类的实现者和使用者都有好处。第一，防止使用者直接操作对象数据时有意无意 造成的错误，毕竟对象自己的方法处理自己的数据才是安全的；第二，使用者通过方法调用 来操作数据时无需了解内部实现细节，类的功能非常易用；第三，即使实现者以后修改了类 的内部实现，只要不改变方法界面，使用者就不会受到任何影响，这使得程序非常容易维护； 第四，可以使同类甚至不同类的对象对使用者都呈现标准化的操作界面。以电视机为例可以 很清楚地看出上述优点。电视机将内部各种器件封装起来，使用户只能通过面板上的按键来 操作电视机，这样既保护了内部器件使之不易被用户损坏，又使电视机简单易用；电视机内 部器件的维修和升级对用户来说没有任何影响；所有品牌的电视机几乎都提供标准化的面板（电源开关、频道切换、音量调节等），只要会使用一种电视机，基本上就会使用其他任何电 视机甚至收音机（因为收音机界面也是类似的调换频道、调节音量等）。

Python 类定义

如前所述，类是用来刻划对象的数据特性和行为特性的。Python 中类定义形如：

```
class <类名>:
    <方法定义 1>:
        ...
    <方法定义 n>:
```

其中诸“方法定义”就是对对象的操作行为的定义，外界能够请求对象所做的事情完全由这 些方法决定。

每个方法定义其实都是一个函数定义，即形如：

```
def <方法名>(<参数>):
    ...
```

方法与普通函数略有差别。首先，每个方法都必须有一个特殊的形参 self，并且 self 必 须是该方法的第一个形参（如果还有其他参数的话）。至于这个特殊参数的作用，我们在稍后 介绍方法调用的时候再解释。其次，方法只能通过对象来调用，即向对象发消息请求对象执 行某个方法。

至此，读者也许会有疑问：不是说类中包含了数据和操作的定义吗，怎么上述类定义形 式中只有操作没有数据？在 Python 中，类的实例所拥有的数据一般在类的方法中定义的，尤 其是在一个特殊方法 init方法①中。这里的方法名 **init**是 Python 规定的，通常每个类 都会定义 init__方法，其功能将在介绍类实例创建时进行解释。

作为例子，我们来定义类 Person。Person 是对现实世界中“人”的抽象，每个人都有自 己的姓名和出生年份，并且都能回答外界有关其姓名和年龄的提问。我们将 Person 类的定义 保存在文件 person.py 中，将来任何程序都可以通过导入此文件而使用 Person 类。

【程序 7.3】person.py

```
class Person:
    def __init__ (self,n,y):
        self.name = n self.year = y
    def whatName(self):
        print "My name is",self.name
    def howOld(self,y):
        age = y – self.year
        if age > 0:
            print "My age in",y,"is",age
        else:
            print "I was born in",self.year
```

Person 对象的数据就是在 init__方法中定义的 self.name 和 self.year，Person 对象能够执 行的操作就是方法 whatName()和 howOld()。

像 Person 类中的 self.name 和 self.year 这样，以“self.<变量名>”形式定义的变量称为实 例变量（instance variable），意思是这种数据是属于类的实例的，不同实例可有不同的值。实 例变量可以在类的任何方法中定义（通常在 init中定义），也可以在类的任何方法中直接 引用。**Person** 类中，**self.name** 和 **self.year** 是实例变量，将来创建的每一个 **Person** 实例都拥有这两个数据，每个实例各自的姓名和出生年份的值可以是不同的；另外，这两个实例变量虽然是在 **init**中定义，但在 whatName 和 howOld 方法中都可以直接引用。注意，如果方法 中定义的某个变量名前没有 self 前缀，则该变量是普通的函数局部变量，其作用域仅限于该方法。例如上例中在 howOld 方法中定义的 age 就是局部变量，它只能在 howOld 中引用。 类的方法定义中可以通过 self.f()的方式调用同一个类中的其它方法 f。例如：

> ① 注意，这个名称中 init 前后各是两个下划线字符！Python 常用这种形式的标识符来命名内部对象。

```
class Person1:
    def __init__ (self,n,y):
        self.name = n
        self.year = y
    def whatName(self):
        print "My name is",self.name
    def howOld(self,y):
        age = y – self.year
        if age > 0:
            print "My age in",y,"is",age
        else:
            print "I was born in",self.year
    def allInfo(self,y):
        self.whatName()
        self.howOld(y)
```

其中的方法 allInfo 调用了本类中的另外两个方法 whatName 和 howOld。

类实际上是用户自定义的数据类型，是对 Python 语言本身提供的基本数据类型的扩充。 一旦定义了类，就可以将它当作 Python 标准数据类型来使用。

作为特例，如果某种对象只包含一些数据，不包含对数据的操作，则可以在类中只定义 实例变量而不定义操作方法（除了特殊的 init 方法）。这样的类相当于很多编程语言中的 “结构”类型①，其作用是将一些数据项组合成一个整体。事实上，类可以看作是传统结构 类型的发展，即类是添加了数据操作的“结构”。

最后说一下命名问题。面向对象编程中习惯上使用大写字母开头的标识符来为类命名（如 Person），而类中方法和实例变量则惯常使用小写字母开头的标识符来命名（如 name、year 和 whatName）。方法名如果由多个单词构成，一般采用骆驼式命名法，即除第一个单词之外 的各单词首字母大写，如 whatName 和 howOld。

7.2.2 对象的创建

一旦定义了类，就可以创建类的实例，也就是该类的一个对象②。类是抽象的，而对象

则是具体的，就好比“人”是抽象概念，而“张三”是个具体的人。一个类可以创建任意多 个实例（对象），所有实例都具有相同的行为（这是由类中定义的方法决定的），但各自的数 据值可以不同。创建类的实例采用如下形式：

```
<变量> = <类名>(<参数>)
```

这里将类名当成一个函数来用，称为类的构造器（constructor，或称构造函数）。构造器返回 一个新对象，通常需要定义<变量>来引用这个新对象。注意，虽然<变量>只是对新对象的引 用，但习惯上我们也常说<变量>就是新对象本身，这一般不会产生混淆。

如果希望创建对象时将对象定制成特定的初始状态，则可以在类中定义特殊的 __init__ 方法③。创建新对象时，Python 自动调用 __init__，实现对新对象的初始化，比如为该对象所 拥有的数据进行赋值。 __init__ 方法可以用参数（至少有一个 self 参数）来传递初始化所需的信息，调用 __init__ 时必须提供相应的实参。但由于 __init__ 不是直接调用的，无法直接将实 参传递给它，所以我们将所需实参传递给构造器，再由构造器自动传递给 __init__ 。不过，__init__ 的特殊参数 self 是一个例外，传递给 self 的实参是新创建的对象（更准确地说是对新 建对象的引用）。

> ① 如 Pascal 语言中的 record 和 C 语言中的 struct 类型。
>
> ② 本书中混用“实例”和“对象”这两个术语，视之为相同的概念。
>
> ③ 也可以说 init ()才是类的构造器，不过不能直接调用，而是通过类名来隐含地调用。

例如，下面的语句先导入 Person 类，然后创建一个 Person 对象，并使变量 p1 引用该对象：

```
>>> from person import Person
>>> p1 = Person("Lucy",2005)
```

创建对象时自动调用 init__方法，该方法所需的三个参数 self、n、y 分别用实参 p1、"Lucy"和 2005 代入，这相当于函数调用

```
__init__ (p1,"Lucy",2005)
```

从而导致执行 __init__的函数体，为新对象进行初始化：

```
p1.name = "Lucy"
p1.year = 2005
```

图 7.6 给出了上述利用 Person 构造器创建对象 p1 并调用 __init__进行初始化的过程。

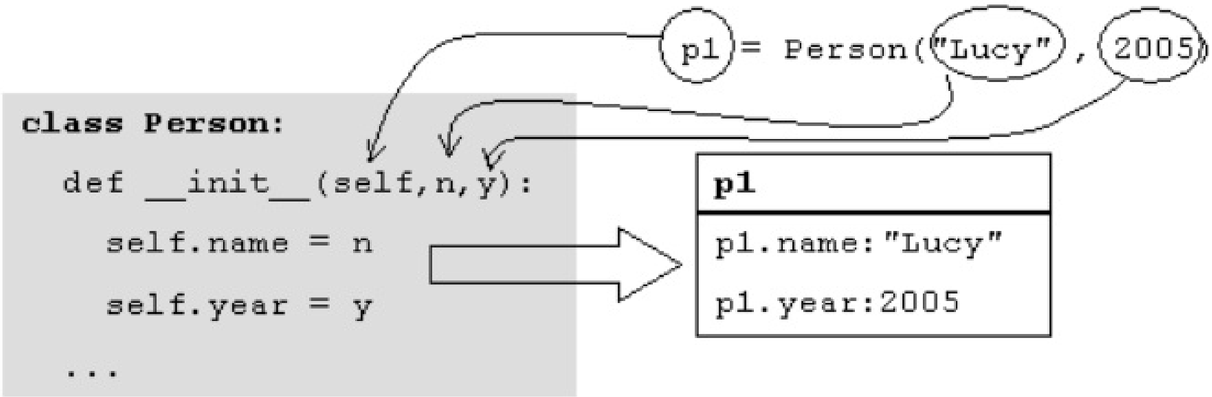


图 7.6 对象创建与初始化

注意， init 方法中对变量 name 和 year 所赋的值"Lucy"和 2005，是专属于新实例 p1 的，它们与同一个类的其他实例（例如下面将创建的 p2）没有关系。这两个变量都属于实例 变量（instance variable），意即它们的值是随实例的不同而不同的。下面再创建一个 Person 对象，并使变量 p2 引用这个新对象：

```
>>> p2 = Person("Tom",1990)
```

同样地，Python 自动调用 init 方法，只不过这次传递给该方法的参数是 p2、"Tom"和 1990，即相当于函数调用

```
__init__ (p2,"Tom",1990)
```

从而导致执行 __init__的函数体，为新对象 p2 进行初始化：

```
p2.name = "Tom"
p2.year = 1990
```

这里，对实例变量 name 和 year 所赋的值"Tom"和 1990 是专属于新实例 p2 的，与前面创建的实例 p1 没有关系。创建同一个类的多个实例的过程可参见图 7.7。

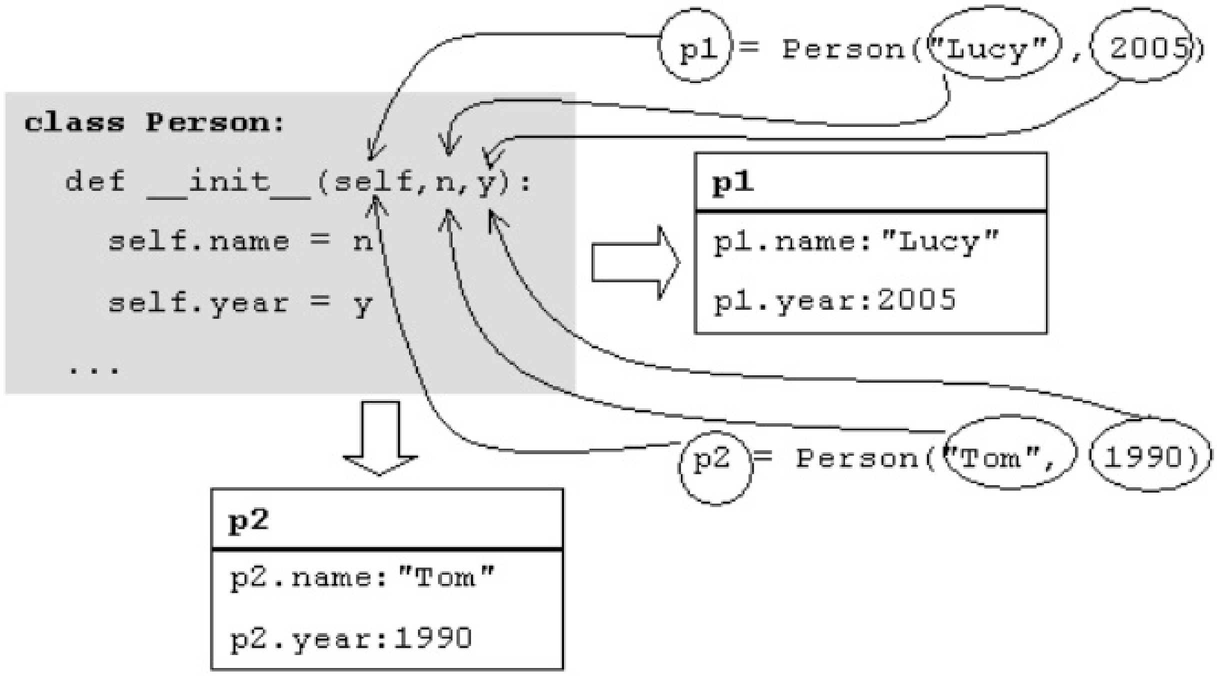


图 7.7 创建同一个类的多个实例

从图 7.7 可见，类与实例的关系就像模具与产品的关系，用同一个模具可以制造出大量 的产品，这些产品总体上是相似的，但可能各有不同的细节。p1 与 p2 是属于同一类的对象， 总体上非常相似，例如都有数据 name 和 year，但各有不同的数据值。

7.2.3 对象方法的调用

一旦创建了对象，就可以通过向对象发消息来调用对象的方法。消息的格式如下：

```
<对象>.<方法>(<实参>)
```

其含义是请求<对象>执行<方法>，方法定义中列出的形式参数由<实参>提供。

例如，接着前面的例子执行如下语句：

```
>>> p1.whatName()
My name is Lucy
>>> p2.whatName()
My name is Tom
>>> p2.howOld(2013)
My age in 2013 is 23
```

前面说过，类中定义的方法都必须以 self 作为第一个参数，这个参数用来指明当前是哪 一个对象实例要执行类的方法。传给 self 的实参就是上述方法调用格式中的<对象>，只不过 这个实参是由 Python 隐含地传递给 self，而不是由程序员在方法调用表示法中直接传递。上 面例子中的 p1.whatName()和 p2.howOld()引起的方法调用过程可参见图 7.8。

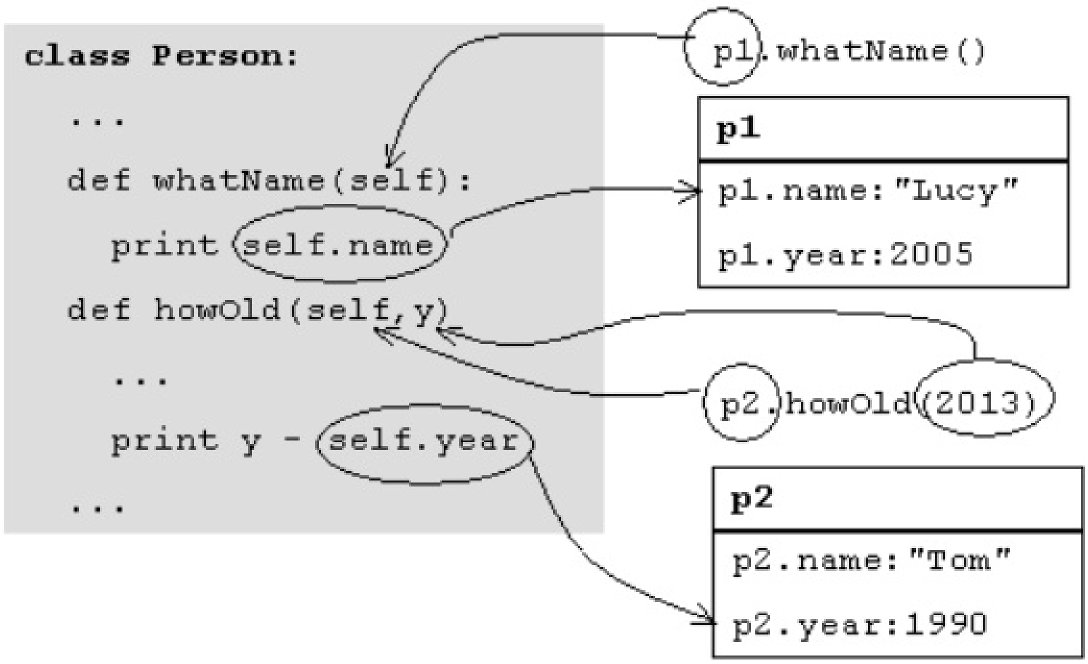


图 7.8 对象方法调用过程

要说明的是，类方法的第一个参数所用的参数名 self 只是 Python 语言的惯例，而非硬性 规定，完全可以使用别的名字①。

7.2.4 编程实例：模拟炮弹飞行

本节讨论一个模拟炮弹飞行的程序的设计。我们采用三种设计方法，得到三个版本的程序。通过比较各个版本的差别，可以看出 OOP 与传统的面向过程编程相比具有明显优点。

算法设计

程序规格是输入炮弹的发射角度、初速度和高度，输出炮弹的射程。虽然可以利用复杂的数学公式直接算出射程，但我们采用模拟炮弹飞行过程的方法来求射程。所谓模拟炮弹飞行过程，就是从炮弹射出炮口开始，计算炮弹在每一时刻的位置（水 平距离和高度），直至炮弹落地。注意，时间和炮弹飞行轨迹都是连续的量，由于计算机不能 处理连续的数值，所以需要将时间和炮弹飞行轨迹“离散化”，也就是将时间划分成一系列离 散的时段，飞行轨迹也相应地划分成一系列离散的点。

炮弹在每一时段所处的位置可以利用简单的中学物理知识求得。将炮弹速度分解成水平 分量和垂直分量，则炮弹在水平方向的运动是匀速直线运动（忽略空气阻力），在垂直方向的 运动是加速运动（因为重力的影响，炮弹先向上减速飞行，减到向上速度为 0 后改为自由落 体运动）。算法伪代码如下：

```
算法：模拟炮弹飞行。
输入：角度 angle（度）、初速度 v（米/秒）、高度 h0（米）、时间间隔 t（秒）
输出：射程（米）
计算初速度分量：先将 angle 换算成弧度单位的 theta，再计算
    xv = v * cos(theta), yv = v * sin(theta)
初始位置：(xpos,ypos) = (0,h0)
当炮弹还未落地（即 ypos &gt;= 0.0）：
    更新炮弹在下一时段的位置(xpos,ypos)和垂直速度分量 yv
输出 xpos
```

> ① Java 和 C++中使用的是“this”。

为了理解此算法，请参看示意图 7.9。

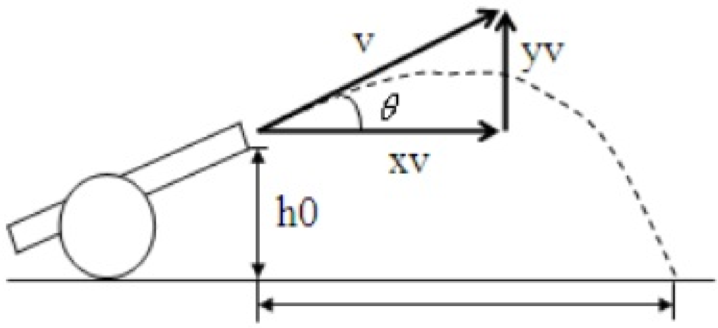


图 7.9 模拟炮弹飞行的有关数据

炮弹飞行过程中，水平位置的更新很简单：按照匀速直线运动的规律，每个时段 t 内，

炮弹都飞行 xv * t 距离，因此炮弹在水平方向从 xpos 运动到了新位置

```
xpos = xpos + xv * t
```

炮弹垂直方向位置的变化稍微复杂点：由于重力的影响，炮弹向上速度每秒减少 9.8 米/ 秒，经过时段 t，向上速度变成了

```
yv1 = yv - 9.8 * t
```

而炮弹在时段 t 内垂直方向位移可以用这段时间的平均速度乘 t 来计算，因为时段 t 内的平均速度为起点速度 yv 与终点速度 yv1 之和的一半，故时段 t 内的垂直方向位移为

```
(yv + yv1) / 2.0 * t
```

于是，经过时段 t 后，炮弹在垂直方向的新位置为

```
ypos = ypos + (yv + yv1) / 2.0 * t
```

最后要说明的是，模拟炮弹飞行的循环语句的条件 y>=0 中之所以用等号，是为了使程 序在初始高度为 h0 = 0 的情况下也能进入循环进行模拟。一旦算出炮弹最新高度小于 0，则 终止循环。

下面是完整程序：

【程序 7.4】cball1.py

```
# -*- coding: cp936 -*-
from math import pi,sin,cos
def main():
    angle = input("输入发射角度(度): ")
    v = input("输入初速度(米/秒): ")
    h0 = input("输入初始高度(米): ")
    t = input("输入时间间隔(秒): ")
    theta = (angle * pi) / 180.0
    xv = v * cos(theta)
    yv = v * sin(theta)
    xpos = 0
    ypos = h0
    while ypos >= 0:
        xpos = xpos + t * xv
        yv1 = yv - t * 9.8
        ypos = ypos + t * (yv + yv1) / 2.0
        yv = yv1
    print "射程: %0.1f 米." % (xpos)
main()
```

以下是程序 7.4 的一次执行结果：

```
输入发射角度(度): 56
输入初速度(米/秒): 300
输入初始高度(米): 2
输入时间间隔(秒): 0.1
射程: 8522.1 米.
```

用写作文打比方的话，程序 7.4 采用的是流水帐式的、毫无章法结构的作文方法，它将所有数据和操作语句全都混在一起。程序虽然不长，却使用了 10 个变量，要想理解这个程序就必须时刻记牢并跟踪这 10 个数据的变化，这对人脑来说是个不小的负担。 模块化程序设计有助于改善程序的结构，增强程序的易理解性。我们利用模块化来重新组织程序 7.3 中的语句，形成一些具有相对独立性的模块（函数）。下面就是炮弹模拟程序的 模块化版本：

【程序 7.5】cball2.py

```
# -*- coding: cp936 -*- from math import pi,sin,cos
def getInputs():
    a = input("输入发射角度(度): ")
    v = input("输入初速度(米/秒): ")
    h = input("输入初始高度(米): ")
    t = input("输入时间间隔(秒): ") return a,v,h,t
def getXY(v,angle):
    theta = (angle * pi) / 180.0
    xv = v * cos(theta)
    yv = v * sin(theta) return xv,yv
def update(t,xpos,ypos,xv,yv): xpos = xpos + t * xv
    yv1 = yv - t * 9.8
    ypos = ypos + t * (yv + yv1) / 2.0
    yv = yv1
    return xpos,ypos,yv
def main():
    angle, v, h0, t = getInputs()
    xv, yv = getXY(v,angle) xpos = 0
    ypos = h0
    while ypos >= 0:
        xpos,ypos,yv = update(t,xpos,ypos,xv,yv)
    print "射程: %0.1f 米." % (xpos)
```

与程序 7.4 相比，程序 7.5 的主程序 main 显得非常简洁、容易理解。main 中用到的变量 从 10 个减到 8 个，少掉的两个变量是 theta 和 yv1。变量 theta 存储的是以弧度为单位的发射角度，它是为了符合 math 库中三角函数的用法而临时创建的中间数据，对程序来说既不是输入数据，又不是输出数据，也不是贯穿算法始终的关键数据。因此，将 theta 隐藏在用到它的函数 getXY 中，是符合它的“跑龙套”身份的做法。基于同样的理由，yv1 也被隐藏在了函 数 update 中。

然而，尽管模块化编程改善了程序的结构，使程序易读易理解，但程序 7.5 的主程序仍 然比较复杂。为了描述炮弹的飞行状态，需要 xpos、ypos、xv 和 yv 等 4 个数据，其中 xpos、ypos 和 yv 是随时间 t 而变的，需要时时更新，这就导致了主循环中的那个复杂、累赘的函数调用：

```
xpos,ypos,yv = update(t,xpos,ypos,xv,yv)
```

函数作为功能黑盒子，应该提供简明易用的接口，而 update 函数的设计显然不够简明易 用，它需要输入 5 个参数，并输出 3 个返回值。这就像一台设计拙劣的电视机，从机壳内伸 出七八根电线，买回家后需要完成复杂的接线之后才能收看电视。请记住，如果函数接口过 于复杂，往往表明这个函数的设计需要改善。

最后，我们用 OOP 来编写炮弹模拟程序。炮弹原本是现实世界中的一个对象，传统编 程方法却用 xpos、ypos、xv 和 yv 等四个分离的数据来描述它，这是典型的"只见树木不见 森林"。假如有一个 Projectile 类来描述炮弹对象，有关炮弹的一切信息和行为都封装在这个 类中，那么在主程序中要做的就是创建一个炮弹对象，然后由这个对象自己完成所有的计算 任务，代码形如：

```
def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print "射程: %0.1f 米." % (cball.getX())
```

这段程序的含义是：首先输入炮弹的初始数据 angle、v、h0 以及计算炮弹飞行位置的时间间隔 t；然后利用这些初始值创建炮弹对象；接着进入主循环，不断请求炮弹更新其位置，直 至炮弹落地。程序中只用到必不可少的 4 个初始数据，其他数据都隐藏在 Projectile 类当中，这使得程序逻辑非常清晰、易理解。

当然，主程序之所以简单，是因为复杂性都被隐藏在类当中了。下面来考虑 Projectile 类 的定义。前面主程序中实际上已经提出了对类的要求，即类中必须实现 update、getX 和 getY 方法。此外，还必须定义类的构造器。

构造器 __init__用于初始化新创建的对象，比如为对象的实例变量赋初值。炮弹对象的实 例变量显然应该包括描述炮弹状态的四个数据：xpos、ypos、xv 和 yv。初始化代码如下：

```
def __init (self, angle, velocity, height):
    self.xpos = 0.0
    self.ypos = height
    theta = pi * angle / 180.0
    self.xv = velocity * cos(theta)
    self.yv = velocity * sin(theta)
```

注意变量 theta 的用途是临时性的，其值只在此处用到，别处不需要，因此没有必要将 theta 也作为炮弹对象的实例变量，而应作为普通的局部变量。

方法 getX 和 getY 很简单，分别返回实例变量 self.xpos 和 self.ypos 的当前值即可。

update 方法是最核心的方法，它的任务是更新炮弹在某个时间间隔后的状态。只需传递 一个时间间隔参数 t 给 update 即可，这比程序 7.5 中的 update 简单多了。代码如下：

```
def update(self,time):
    self.xpos = self.xpos + time * self.xv
    yv1 = self.yv - time * 9.8
    self.yp = self.yp + t * (self.yv + yv1)/2.0
    self.yv = yv1
```

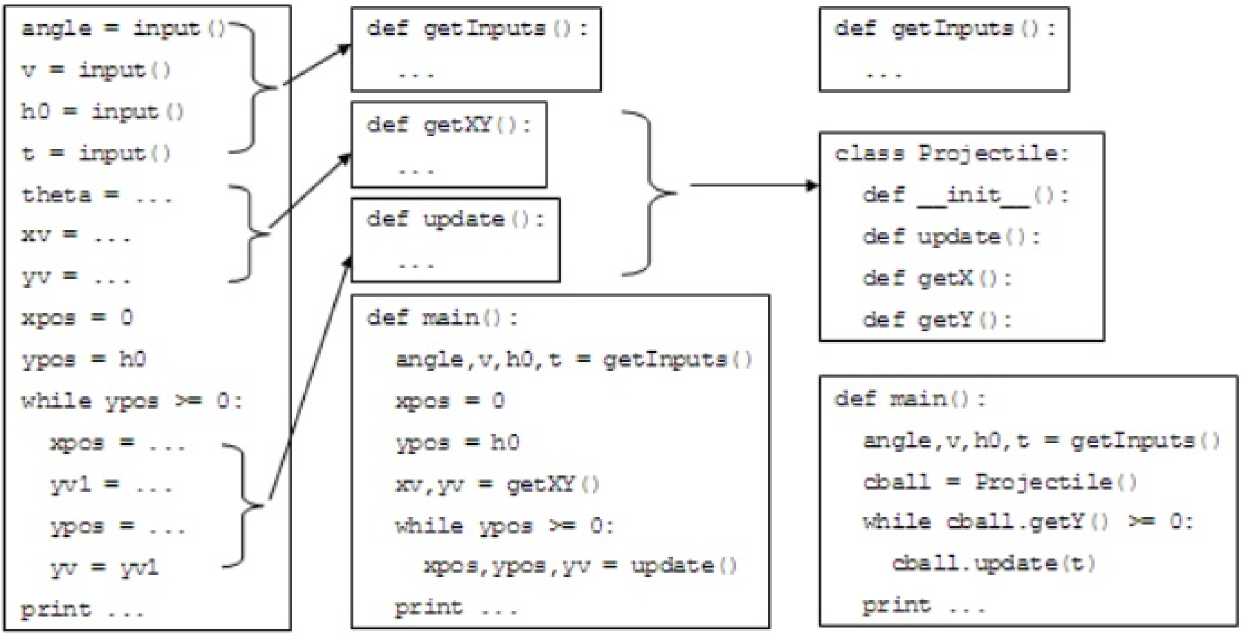
注意 yv1 也是一个普通的临时变量，它的值在下一次循环中就是 yv 的值，因此程序中将其值保存到实例变量 self.yv 中。

至此，我们就完成了 Projectile 类的定义。再添加 getInputs 函数后，就得到完整的面向对象版本的炮弹模拟程序。

【程序 7.6】cball3.py

```
from math import pi,sin,cos
class Projectile:
    def __init__ (self,angle,velocity,height):
        self.xpos = 0.0
        self.ypos = height
        theta = pi * angle / 180.0
        self.xv = velocity * cos(theta)
        self.yv = velocity * sin(theta)
    def update(self, time):
        self.xpos = self.xpos + time *
        self.xv yv1 = self.yv - 9.8 * time
        self.ypos = self.ypos + time * (self.yv + yv1) / 2.0
        self.yv = yv1
    def getX(self):
        return self.xpos
    def getY(self):
        return self.ypos
def getInputs():
    a = input("输入发射角度(度): ")
    v = input("输入初速度(米/秒): ")
    h = input("输入初始高度(米): ")
    t = input("输入时间间隔(秒): ") return a,v,h,t
def main():
    angle,v,h0,t = getInputs()
    cball = Projectile(angle,v,h0)
    while cball.getY() >= 0:
        cball.update(t)
    print "射程: %0.1f 米." % (cball.getX())
```

本程序三种版本的设计思想变迁，可以用图 7.10 来刻划。



(a) 非模块化过程 (b) 模块化 （c）面向对象

图 7.10 炮弹模拟程序不同设计方法的变迁

7.2.5 类与模块化

我们在第 4 章讨论过模块化编程的思想。对于复杂程序，通常需要用分解的方法将程序 划分成若干模块，使每个模块仅针对有限的数据执行有限的操作。模块化能够使复杂程序的 设计更加可控。

对复杂程序一般有两种分解方法：功能分解和数据分解。功能分解是面向过程编程的基 础，依赖于子程序（如函数）概念，以过程为中心来建立功能模块；数据分解则是面向对象 编程的基础，依赖于类的概念，以数据为中心来建立数据模块。

功能模块不太适合复杂数据的处理。以处理“学生”数据的程序为例，如果按功能分解， 需要建立课程注册模块、修改学生信息模块、成绩登录模块等等。每一个模块（函数）的编 写，都需要知道“学生”数据的各种细节。

而数据模块则可以避免功能模块的不足。通过对“学生”的数据和操作的抽象，创建学 生类 S，对学生数据能够执行的操作构成 S 的对外界面，而操作的实现细节则隐藏在 S 内部， 从而使 S 的使用者无需了解“学生”数据的细节就能执行所需操作。

两种模块化方法具有类似的优点，如代码重用、易维护、支持团队开发等，但他们导致 的程序具有完全不同的执行方式。面向对象程序是由很多对象组成的，对象之间通过交互（发送、接收消息）、协作完成计算任务，而传统程序则是由一系列预定的过程组成的，通过按顺序执行这些过程而完成计算任务。 模块化设计体现了信息隐藏的思想，即程序模块应当对模块用户尽可能隐藏内部细节，只保留必要的访问界面。对功能模块（函数），以 math 库中的函数 sqrt()为例，我们作为调用 者，并不知道该函数的内部实现细节，如数值的表示细节和求平方根的算法细节，而只需要 知道该函数能够对给定的数值求平方根即可。对数据模块（类）同样如此，以程序 7.6 定义 的 Projectile 类为例，该类的使用者无需了解炮弹究竟用什么数据来表示以及如何计算其飞 行，只需要了解该类的使用界面（update、getX、getY）就能编写炮弹模拟程序。

既然类是一种具有独立性的程序模块，就可以单独存储在模块文件中，无需与使用类的 代码（主程序）存储在一个程序文件中。这样做的好处是类模块可以重用，任何想使用这个 类的程序都可以导入类模块。例如，我们可以将 Projectile 类定义单独保存在模块 proj.py 中， 任何希望使用 Projectile 类的程序只需导入它，导入后即可创建对象、执行对象方法。就像下面这样：

```
from proj import Projectile
def main():
    angle, vel, h0, time = getInputs()
    cball = Projectile(angle, vel, h0)
    while cball.getY() >= 0:
        cball.update(time)
    print "射程: %0.1f 米." % (cball.getX())
```

我们当然可以让每个类单独构成一个模块，但这样一来，当类的数目很多时会导致模块 数目过多，反而增加程序的复杂性。实际上我们通常是将若干个相关的类存储在一个模块文 件中，例如 5.4.2 节介绍的 graphics.py 模块中就包含了所有图形类。不过，使用类的程序一 般都放在与类模块不同的模块中。

很多面向对象编程语言都以“类库”的形式提供具有各种实用功能的类模块给程序员使 用，就像过去面向过程编程语言提供“函数库”一样。OOP 往往能非常简单地解决复杂问题， 因为专业的程序员已经开发了大量可重用的代码。

7.2.6 对象的集合体

一个复杂数据之“复杂”主要体现在两个方面：要么该数据是由大量成员数据组成的，要么该数据具有深层的内部结构。第 6 章介绍了如何利用各种集合体数据类型和数据结构来 表示数量复杂性，本章介绍的类则可以刻画内部结构的复杂性。

可以推想，如果将集合体类型与类相结合，就能表示现实中的任意复杂的信息。即，用 集合体表示大量的数据成员，而每个数据成员是一个具有复杂内部结构的对象。我们不妨用 下面的公式来表达这个思想：

```
类 + 集合体 = 任意复杂的数据
```

例如，如果程序中要处理的数据是“一群人”，那么我们可以利用一个 Person 对象的列表来表示这一群人。假设我们已经创建了若干个 Person 对象，如 7.2.2 中创建的 p1 和 p2，下 面的代码将这两个对象存储在一个列表 people 中。现在 people 就是一个非常复杂的数据，既 有大量的成员，而且每个成员本身又是复杂的对象。我们可以利用循环语句对复杂数据 people 中的所有成员进行特定处理（如显示姓名和年龄）：

```
>>> people = [p1, p2]
>>> for p in people:
        p.whatName()
        p.howOld(2013)
My name is Lucy
My age in 2013 : 8 My name is Tom
My age in 2013 : 23
```

7.3 超类与子类*

当我们用类去描述现实世界的对象时，有时会发现某些类之间是“一般与特殊”的关系。

例如，“人”与“学生”之间就是一般与特殊的关系，而“学生”与“研究生”也是一般与特 殊的关系。当然，“人”的特殊例子还包括“教师”，“学生”的特殊例子还包括“旁听生”。 总之，通过一般与特殊的关系，可以将所有类组织成为一个层次结构，称为类层次。如图 7.11 所示。

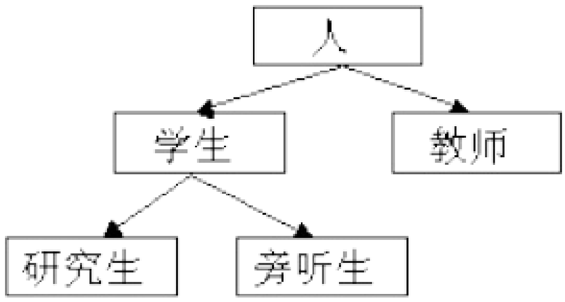


图 7.11 类层次

为了描述这种一般与特殊的关系，面向对象语言中提供了相应的类定义方式。具有一般性的类称为超类（superclass），具有特殊性的类称为子类（subclass）①。例如在图 7.11 中，“人”是“学生”的超类，“学生”是“人”的子类；“人”也是“教师”的超类，“教师”是 “人”的子类；“学生”又是“研究生”的超类，“研究生”是“学生”的子类；等等。

7.3.1 继承

不难理解，子类拥有超类的一切特性，凡是超类适用的地方，子类也适用。例如，“研究生”具有“学生”的全部属性，包括数据属性（如学号、姓名、年龄）和行为属性（如选课、 参加学生社团等），凡是“学生”能做的，“研究生”都能做。子类拥有超类的全部属性（数 据和方法），这是面向对象方法中极为重要的一个特色，称为继承（inheritance）。

子类除了继承超类的属性，还包含一些自己的特殊属性。例如，“研究生”具有导师信 息，而一般的“学生”未必有导师。在进行面向对象设计时，一般先定义超类，然后在超类 基础上通过添加一些特殊属性来定义子类。这种定义方式下，子类中不必重复定义那些继承 来的属性，从而简化了子类定义。这也是继承机制带来了的另一个重要特色——代码重用（code reuse），即超类中的代码可以通过继承机制被子类重复使用。当我们需要定义一个新 类时，如果发现它与某个现有的类在很多方面都相同，那么就无需重新写代码来实现这些相 同行为，而只需继承现有功能。

> ① 超类/子类也称为基类（base class）/派生类（derived class）。

Python 中定义子类采用如下形式：

```
class <子类名>(<超类名>):
    <特殊属性 1>
    ...
    <特殊属性 n>
```

注意，超类与子类的定义一般置于同一个模块中。如果超类在另一个模块中定义，则定义子类时必须指明模块信息，形如：

```
class <子类名>(<模块名>.<超类名>):
    ...
```

下面通过具体例子来说明超类、子类以及继承概念。程序 7.3 中定义了一个 Person 类， 它在最一般的层次上刻划了“人”对象：每个人有姓名和出生年份数据，并且能回答外界提 出的“叫什么名字”、“今年多大了”之类的问题。为便于阅读、比较，我们将 Person 类的定 义复制于此：

```
class Person:
def __init__ (self,n,y):
    self.name = n
    self.year = y
def whatName(self):
    print "My name is",self.name
def howOld(self,y):
    age = y – self.year
    if age > 0:
        print "My age in",y,"is",age
    else:
        print "I was born in",self.year
```

下面以 Person 作为超类，来定义几种具有特殊属性（数据和行为）的人。 首先定义“学生”。学生是人，因此拥有 Person 类的所有信息。学生还具有一些特殊属性，比如学校、学号信息。按照子类的定义方式，我们可以定义如下的 Student 类 :

```
class Student(Person):
    def __init__ (self,n,y,u,id):
        Person.__init__ (self,n,y)
        self.univ = u
        self.snum = id
    def getUniv(self):
        return self.univ
    def getNum(self):
        return self.snum
```

Student 类定义的第一行表明，Student 类是 Person 类的子类。作为特殊的人，学生拥有 普通人不一定有的 self.univ（学校）和 self.snum（学号）数据，因此 Student 类的构造器与 Person 不同，需要四个初始参数：姓名、出生年份、学校和学号。创建 Student 对象时要进行的初始化工作包括：首先作为 Person 对象要执行 Person 对象的初始化，即 Person.__init__()；然后再执行 Student 对象特有的初始化工作，即对 self.univ 和 self.snum 两个实例变量进行赋 值。可见，子类的构造器一般是在超类构造器的基础上另外执行一些初始化工作。Student 对象除了能响应 Person 对象都能响应的 whatName 和 howOld 之外，还具有两个特有的方法： getUniv 和 getNum。 我们再来定义另一种特殊的人——教师。假设教师拥有指导的学生人数信息，以及设置和获取这个信息的方法，则可定义 Teacher 类如下：

```
class Teacher(Person):
    def setNum(self,n):
        self.snum = n
    def getNum(self):
        return self.snum
```

Teacher 类没有定义自己的初始化方法 init，因此创建 **Teacher** 对象时将自动调用超类 **Person** 的 **init**方法来进行初始化。Teacher 对象有一个特殊的实例变量 num，但其值不是 在创建对象时初始化的，而是在创建之后利用 setNum 方法来设置。不要忘了，Python 对象 的实例变量可以在任何方法中定义，可以在任何时候赋值。

定义了以上各种类之后，就可以编写使用这些类的程序了。假设 Person、Student 和 Teacher 类定义都存储在模块文件 person.py 之中，下面以交互方式演示对这些类的使用：

```
>>> from person import *
>>> tom = Student("Tom",1995,"SJTU","S001")
>>> tom.whatName()
My name is Tom
>>> tom.howOld(2013)
My age in 2013 is 18
>>> tom.getUniv()
'SJTU'
>>> print tom.getNum()
S001
>>> huck = Teacher("Huck",1975)
>>> huck.whatName()
My name is Huck
>>> huck.howOld(2013)
My age in 2013 is 38
>>> huck.setNum(8)
>>> print huck.getNum()
8
>>> lucy = Person("Lucy",2005)
>>> lucy.getUniv()
Traceback (most recent call last):
File "&lt;pyshell#4&gt;", line 1, in &lt;module&gt; p.getUniv()
AttributeError: Person instance has no attribute 'getUniv'
```

子类继承超类的所有属性，因此当创建了 Student 对象 tom 后，就可以向 tom 发消息 whatName 和 howOld，tom 对象能够正确地响应这两个消息。当然还可以向 tom 发送 getUniv 和 getNum 消息，这两个方法是 Student 特有的，tom 自然能做出响应。Teacher 对象 huck 的行为也是类似的。注意，超类的实例并不具有子类中特殊属性，因此上例中向 Person 对象 lucy发送 getUniv 消息，将导致错误。

7.3.2 覆写

子类除了原样继承超类的方法，还可以修改继承来的超类方法，以便以自己的方式行事。 这种在子类中重新定义超类方法的情况是面向对象的又一特色，称为覆写（override，或称重 定义）。

例如，我们来定义另一类特殊的人——娱乐圈明星。娱乐圈明星当然是人，所以他们都 具有 Person 的属性。但明星们一般都很忌讳被问年龄，他们才不会像普通人那样直接回答 howOld 问题。因此，我们在定义 Star 类时需要重新实现 howOld 方法，不能直接使用 Person 中定义的 howOld 的代码。Star 定义如下：

```
class Star(Person):
def howOld(self,y):
    print "You guess?"
def setYear(self,y):
    self.year = y
```

类 Star 中重新定义了 howOld()方法，将来向 Star 对象发送 howOld 信息时，它就会执行自己 独特的 howOld 方法。当然，如果向 Star 对象发送 whatName 消息，由于明星们在这个问题 上与常人无异，该对象就会去执行原样继承来的超类 Person 中的 whatName 方法。顺便指出， 上述 Star 类定义中还定义了一个特殊方法 setYear，这是为了满足某些明星直接修改自己出生 年份的需求:-)，同时也是为了说明子类既可以覆写从超类继承来的行为，也可以干脆定义新 的行为。下面是 Star 类的使用例子（假设 Star 类定义已经导入）：

```
>>> liu = Star("Liu",1955)
>>> liu.whatName()
My name is Liu
>>> liu.howOld(2013)
You guess?
```

注意，覆写是指在子类中重新实现超类的方法，该方法的调用界面（参数和返回值）是 不能改变的。另外，子类中覆写的方法仅适用于子类对象，并不能取代超类中的对应方法。 还用上述例子，当向 Person 类的实例发送 howOld 消息时，仍会执行原来的 howOld 代码。 有趣的是，由于 Star 对象也是 person 对象，我们甚至能强迫 Star 对象执行 Person 中的 howOld 方法来如实回答年龄，做法如下（参见图 7.8）：

```
>>> Person.howOld(liu,2013)
My age in 2013 is 58
```

7.3.3 多态性

在 7.3.1 中定义的类 Student 和 Teacher 中，有一个同名的方法 getNum。虽然同名，但这个方法在两个类中的行为是完全不同的：在 Student 中返回的是学号，而在 Teacher 中返回的 是学生人数。因此，当我们向一个对象 obj 发送 getNum 消息时，所得结果取决于 obj 的类型。 在 OOP 中，多个不同类的对象都支持相同的消息，但各对象响应消息的行为不同，这种能 力称为多态性（polymorphism），即同一操作具有不同形态的意思。

其实多态性对我们来说并非很陌生的概念。考虑表达式“a+b”，请问这个表达式执行什么操作？答案是有多种可能。如果 a、b 是数值，则该表达式执行数值加法运算；如果 a、b 是字符串，则该表达式执行字符串合并操作；如果 a、b 是两个二元组，则该表达式执行结果 是一个四元组；等等。像这样，一个操作的含义依赖于被操作的对象，就是多态性。

多态性使得我们能够刻划不同类所提供的相似方法，对使用者来说易理解、易使用，能 够减少编程错误。相反，不同类的相似方法如果定义为不同名字，对使用者来说就很不方便。 例如，在 Windows 环境下，“双击”就是一个多态操作，双击不同对象导致的行为是不同的。 双击可执行文件，能够执行程序；双击 mp3 文件，可以播放音乐；双击窗口标题栏，可以极 大化或恢复窗口大小；等等。用户如果知道“双击”大体上执行“打开”这个动作的话，那 么学习使用 Windows 时就能举一反三。

多态性的一种典型用法是，让处于同一层次的多种对象都能响应同一个消息，但导致的 行为由各对象决定。例如，如果“人”有学生、教师、官员等子类，这些子类就是处于同一 层次的，假设他们都能响应消息 getNum：学生返回自己的学号，教师返回学生人数，官员返 回工资数额。尽管对象行为各不相同，但在编程时我们可以不管这些差别，以一种统一的方 式来处理他们。假设 tom 是个学生，huck 是个教师，jerry 是个官员，则我们可能写出下列代码来统一处理这些对象：

```
>>> people = [tom,huck,jerry]
>>> for p in people:
print p.getNum()
```

7.4 面向对象设计*

理解了面 向 对象的基 本 概念之后， 就可以应 用 这些概念 来 进行面向 对 象 设 计（object-oriented design，简称 OOD）。 传统的程序设计方法是结构化的自顶向下设计，其思想是将软件系统分解为若干个功能，

每个功能都是对数据的一个操作过程。功能又可以划分为若干个子功能，如此反复分解下去， 直至每个功能所对应的操作过程显而易见为止。在分解功能的同时，还要考虑各功能之间的 接口。这种方法是以过程（函数）为中心的，每个函数都是一个黑盒子，函数调用者只需了 解函数的功能，无需知道实现功能的细节。

面向对象设计是以数据为中心来展开的。对于某种客观实体的数据，考虑可能施加在数 据上的各种操作，然后将数据和操作封装成一个黑盒子——对象。对象通过界面向外界提供 数据操作服务，服务的使用者只需了解服务接口，不必关心服务的实现细节。即使改动了对 象内部的实现细节，只要服务接口不变，所有使用该服务的程序代码就不需要改变。同样地， 对象作为服务提供者，也不需要考虑它的服务将被使用者如何使用，只需确保其服务能正确 处理数据即可。

因此，OOD 的中心任务就是设计各种对象，将对象的数据和行为用类定义出来。OOD 将一个复杂问题分解成一组相互协作的类，以类为设计单位可以大大减小设计的复杂性。

下面是 OOD 的一些指导准则。

描述问题

面向对象技术专家 G. Booch 提出了一种基于词性分析的设计方法，该方法要求设计人员 从目标系统的描述出发，通过问题描述中的名词和动词，来发现候选对象及对象方法。因此，OOD 的第一步是要建立待解决问题的准确、无二义性的描述。问题描述应该是自然、客观的， 不要加入人工的、主观的因素，这是因为面向对象思想的宗旨就是按客观世界的本来面目来 建模并开发应用系统。

找出候选对象

我们的目标是找出有助于解决问题的对象。从问题描述入手，找出问题描述中的名词， 然后逐个考察这些名词，看看是否合适作为对象。对象一般都包含一组相关数据项，如学生（包含学号、姓名、年龄等数据项）、课程（包含课程名、学分、教材等数据项）。而能用单 一数据表示的，或者只有单一行为的实体，一般不适合作为对象，如人数、姓名等。

注意，由于人类可以同时考虑和理解的问题数目受到大脑记忆和处理能力的制约，因此 认定的对象数目不宜过多。合适的对象通常是问题中自然出现的而非人工生硬构造的实体， 而且对象应该向外界提供足够复杂的服务。

确定对象的数据属性

对于认定的对象，接下来就该确定对象的数据。能确定为对象数据的信息一般都具有普遍性，即所有同类对象都拥有的数据，例如学生的学号和姓名。另外，对象数据必须对解决 问题有用，例如，学生的学号、姓名都是信息管理应用中必需的信息，而学生的发型可能就 与应用无关。注意，对象的数据可能是学号、姓名这样的基本数据类型值，也有可能是复杂 类型的值，甚至可能是另一种对象。例如，学生选修的课程也是属于学生对象的数据，课程 本身也是对象。

确定对象的行为属性

认定了对象及其数据之后，接着考虑对象需要什么操作。我们从问题描述中找出动词，它们通常描述了对象的行为。例如，“学生选修课程”中的“选修”就是学生对象的行为。对 象的方法通常使用动词来命名。

一类常见的方法是对实例变量的读取和设置方法。假设对象有实例变量 value，则通常应 提供 getValue 和 setValue 方法。这是因为对象数据是隐藏的，外界只能通过对象的方法来操作对象数据。

对象的方法就是对象向外界提供的界面。界面不完善（如缺少某些方法）会使对象的使 用者无从完成工作，但也不是说向外提供的方法越多越好。对象的界面设计应当遵循恰好够 用的原则，即在能满足用户需要的条件下，界面越简洁越好。

实现对象的方法

有些方法用寥寥数行代码就能实现，有些方法则可能需要设计复杂的算法来实现。对于复杂方法，可以利用自顶向下逐步求精方法来设计。 在方法实现过程中，可能发现一个对象与其他对象之间的新的交互，这会提示我们为类增加新方法，或者增加新的类。

迭代设计

对于复杂程序设计，没有人能够一次性地顺利走过设计全过程。在设计过程中，经常需 要在设计、测试、增加新类或为现有类增加新方法等步骤之间循环往复。

应当多尝试替代方案，即使一个设计方案看上去不太灵，也可以沿着该方案的方向试着 前行，看看会导致什么结果。好的设计一定是通过大量试错而得到的，正是因为错误的设计 才使我们明白应该设计什么样的系统。

最后要指出，上述基于名词动词分析的 OOD 方法只是一个简单的策略，真正进行类的 设计时情况往往很复杂。究竟是否应当设计某个类并没有绝对的设计准则，经常依赖于设计 者的经验。和任何别的设计一样，程序设计既是艺术也是经验，可以通过不断的实践来掌握设计方法。

还要指出，对于小程序，OOD 一般起不了什么作用，说不定反而使编程变得麻烦。但当 编写的程序越来越大，类和对象就能很好地组织程序，减少代码量。

7.5 练习

1. 比较关于数据和操作的两种观点。

2. 什么是封装？

3. 类中方法 init 的作用是什么？

4. 类中方法定义的第一个参数为什么很特殊？

5. 创建类的实例的过程是怎样的？

6. 解释实例变量与普通函数局部变量的异同。

7. 为什么对象集合体能表示任意复杂的数据？

8. 创建交通工具类，以及汽车、飞机子类。

9. 读下列代码，给出其执行结果。

```
class Toy:
    def __init__ (self, value):
        print "Creating a Toy from:", value self.value = 2 * value
    def play(self, x):
        print "Playing:", x
        print x * self.value
        return x + self.value
def main():
    print "Playing around now."
    t1 = Toy(3)
    t2 = Toy(4)
    print t1.play(3)
    print t2.play(t1.play(2))
main()
```

10. 设计实现 Card 类和 Deck 类，Card 实例是一张扑克牌，Deck 实例是一副扑克牌。这两个类应该提供诸如洗牌、发牌等方法。编写主程序来使用这两个类。

第 8 章 图形用户界面

随着 Windows 之类的图形化操作系统的产生和发展，如今用户在与计算机打交道时基本 上都使用形象直观、简单易学的图形化方式，即通过鼠标点击菜单、命令按钮等图形化元素 来向应用程序发出命令，而应用程序也以消息框、对话框等图形化元素来向用户显示各种信 息。因此，为程序建立图形化的用户界面已经成为当今程序设计必备的基本技术之一。本章 介绍图形用户界面的设计和实现，具体内容包括图形构件的用法、Python 标准 GUI 工具包 Tkinter、事件驱动编程以及模型－视图(MV)设计方法。

8.1 图形用户界面概述

8.1.1 程序的用户界面

界面是指两个体系之间的分界与接合部分，例如人－机界面、水－油界面等。在程序设计领域，一个程序的用户界面（user interface，简称 UI）指的是程序中与用户进行交互的部 分，用户通过 UI 向程序输入数据或者请求程序执行特定任务，而程序通过 UI 向用户显示各 种信息。

如果程序员写的程序是自用的，那么用户界面是怎样的并不重要，因为程序员完全了解 程序的行为，能够以最直接的方式来控制程序的运行。但实际上程序员往往是在为其他用户 写应用程序，而用户并不了解程序的内部行为，甚至对计算机技术也可能只是一知半解，因 此程序员必须为应用程序设计用户友好的（user friendly）界面，以便用户能很好地与应用程 序交互。所谓“用户友好”并没有严格的定义，大体指界面易学易记，用户能够高效率地与 计算机进行交互，交互过程中不易犯错，即使犯错也容易恢复。

在本章之前，我们写的程序都是所谓控制台程序，这种程序一般只提供命令行界面（Command Line Interface，简称 CLI），即用户通过键盘输入文本数据或文本命令来控制程序 的行为，而程序向用户显示的也都是文本信息。

与命令行界面不同，图形用户界面（Graphical User Interface，简称 GUI①）提供图形化 界面来实现程序与用户的交互。在 GUI 中，用户通过直接操作窗口、菜单、按钮等图形元素 来向程序发出命令或输入数据，而程序通过消息框、对话框等图形元素来向用户显示信息。 由于图形界面非常直观、易理解，所以 GUI 使得只具有一点基本计算机技能的用户也能顺利 地与计算机打交道。

作为例子，图 8.1 展示了读者已经熟悉的两种界面的 Python 解释器程序：命令行界面和 GUI（即 IDLE）。相信读者已经体会到 IDLE 在编辑源代码、运行和调试程序时的便利和高效。





图 8.1 Python 解释器的两种用户界面

> ① GUI 可读作[gu:i]。

通过操作系统的演化史也可以清楚地了解两种界面的优劣。操作系统是计算机上最重要 的系统软件，用户通过操作系统提供的命令来使用计算机。早期的计算机都使用命令行界面 的操作系统，典型的如 DOS 和 UNIX。用过 DOS 的人都知道，为了让计算机做事情，需要 记忆很多 DOS 命令。例如为了将文件 myfile.txt 从 d:\拷贝到 d:\mydir 目录中，需要输入如下 命令：

```
C:\> copy d:\myfile.txt d:\mydir
```

为了让计算机更加易用，后来人们发明了图形界面的操作系统，典型的如 Microsoft Windows 和 X Window。在 Windows 中要想做上面这条 DOS 命令所做的事情，根本不需要键 盘，只需用鼠标点击几下进行复制粘贴，甚至直接拖动文件到新的文件夹即可。自从有了 Windows，今天的计算机用户可能都不知道曾经有 DOS 这样的东西了。

总之，GUI 能够大大增强程序的用户友好性，提高用户使用计算机的效率，因此是程序 设计中的重要技术。

8.1.2 图形界面的组成

应用程序的图形界面是由底层操作系统支持的，不同操作系统平台的图形界面风格不尽相同，但组成界面的图形元素都是类似的。下面我们采用 Python 的标准图形界面工具包Tkinter 的术语来介绍图形界面元素。 图形界面由多种图形元素组成，这些图形元素称为构件（widget）①。就如一部机器由各种零部件组成一样，图形界面这部“机器”的零部件就是构件。从程序角度看，每个构件都 表示了程序的某个数据并提供了对此数据的操作。用户与构件进行交互，从而使用或控制程 序的某个数据。设计 GUI 时，程序员的任务就是合理地利用各种构件，将它们搭配组合起来，， 目标是提高用户与应用程序的交互效率。

窗口（window）可能是读者最熟悉的一种 GUI 构件了②，它是一个由程序控制的矩形屏 幕区域，在此区域中可以放置其他构件。像窗口这样的能够容纳其他构件的构件，一般称为 容器（container）。对于窗口，用户常做的操作是移动、改变大小等。每个 Tkinter 程序都必 须创建一个最外层的窗口，称为根窗口。

在窗口中通常会布置许多用于与用户进行交互的构件，如标签（label）、按钮（button）、菜单（menu）等等。这些构件是基本界面元素，它们不再包含其他构件。每种构件都有各自的用途，例如接受用户输入、执行命令、显示信息等。

如果窗口中包含的构件很多，布置不当的话会使界面杂乱无章，降低界面的易用性。这 时可以用框架（frame）构件来分隔窗口空间和组合构件，以使界面结构清晰。框架也是矩形 屏幕区域，通常用作容器，是建立多级结构的图形界面的基本组织工具。例如，图 8.2 显示 的窗口中用到两个框架和两个按钮，框架 F1 中包含两个勾选钮。

> ① 别的系统也有称为控件或组件的。
>
> ② 微软公司的图形化操作系统干脆以 Windows 命名。

图 8.2 多级结构的界面

窗口或框架是容器构件，能够包含其他构件，由此可见构件之间存在着一种层次关系， 称为父子关系。在图 8.2 中，窗口 W 包含框架 F1 等构件，我们称 W 是 F1 等构件的父构件， F1 等都是 W 的子构件。同样，F1 又包含勾选钮 C1 和 C2，故 F1 是 C1 和 C2 的父构件，C1 和 C2 是 F1 的子构件。设计 GUI 时，必须明确指出构件之间的父子关系。仍以图 8.2 为例：创建 F1 时必须指明是在 W 中创建，创建 C1 时必须指明是在 F1 中创建。

按照构件之间的父子关系，一个图形界面中的所有构件形成了一个树状层次结构，该层 次结构的最高层是根窗口。任何构件（根窗口除外）必有唯一的父构件，还可能有若干子构 件。

在父构件（窗口或框架）中如何安排子构件的位置？这属于图形界面的布局问题，GUI 工具包一般提供布局管理器（layout manager 或 geometry manager）用于布局设计。Tkinter 提供了 Pack、Grid 和 Place 等三种布局管理器，使得在容器中布置子构件的任务轻而易举。

8.1.3 事件驱动

图形构件组成了图形界面的可见部分，在这些可见构件的背后，还有不可见的程序逻辑。 就好比家用电器都提供操作面板，用户通过操作面板控制、使用电器功能，在面板的背后是 实现功能的电路逻辑。

GUI 应用程序的特点是注重与用户的交互，因此程序的执行取决于与用户的实时交互情 况。例如 Word 程序启动后，并非一路执行到程序结束，而是在做了必要的初始化工作后就 停下来，等待用户的下一步动作。用户可能在文档窗口中输入文本，也可能通过菜单设置选 项，还可能点击工具栏里的存盘图标，总之是完全不确定的。Word 程序只能等到用户的交互 动作发生后，才去执行相应的处理代码。

由于 GUI 程序的执行流程由用户控制，并且不可预期，为了适应这种特点，我们需要采 用事件驱动的编程方法。普通程序的执行可概括为“启动——做事——终止”，而事件驱动的 程序的执行可概括为“启动——事件循环（即等待事件发生并处理之）”。作为特例，GUI 程 序的终止也是由特定事件（如关闭窗口事件）引起的。

事件（event）是针对应用程序所发生的事情，并且应用程序需要对这种事情做出响应。 程序对事件的响应其实就是调用预先编制好的代码来对事件进行处理，这种代码称为事件处 理程序（event handler）。GUI 中最常见的事件是用户的交互动作，如按下某个键或者点击鼠标。当然在其他类型的应用程序中也会出现其他类型的事件，例如在各种监控系统中，传感器采集环境数据并传给程序，就可视为发生了需要处理的事件。又如在面向对象程序中，向某个对象发送消息，也可看成是发生了某种需要响应的事件。事件驱动编程（event-driven programming）就是针对这种“程序的执行由事件决定”的应用的一种编程范型。

事件驱动的程序一般都有一个主循环（main loop）或称事件循环，该循环不停地做两件 事：事件监测和事件处理。首先要监测是否发生了事件，如果有事件发生则调用相应的事件 处理程序，处理完毕再继续监测新事件。那么，主循环如何监测事件以及如何触发相应的事 件处理程序呢？这个问题牵涉到操作系统的低层机制，比较复杂。好在这部分代码是独立于 具体应用程序的，一般都由 GUI 工具包提供支持，应用程序员只需编写自己的事件处理程序。

8.2 GUI 编程

8.2.1 UI 编程概述

编写 GUI 程序与编写控制台程序既有相似点，又有一些差别。一方面，任何程序都要利 用编程语言的顺序、循环、分支、函数、模块等成分来搭建程序总体结构、控制程序流程；

另一方面，控制台程序要实现的功能一般都没有现成代码，需要程序员自己编制，而 GUI 程序中的界面设计有 GUI 工具包支持，程序员的编程工作可以大大减少。这是因为图形界面在技术上涉及很多底层细节，在功能上又具有与特定应用无关的通用性，所以很适合由专业的软件厂商来实现，并以工具包的形式提供给程序员使用。

针对不同的操作系统平台、不同的编程语言，存在不同的 GUI 工具包。每种工具包都有 自己的编程界面和程序设计模式，程序员必须学习并遵循这些模式。有些工具包可以运行在 多种操作系统（如 Windows，Unix，MacOS）之上，并能在多种编程语言中使用，称为跨平 台的工具包。程序员一般都固定使用某种跨平台工具包，而不是换个平台就换个工具包，因 为学习使用一个新的工具包可能比学习一个新的编程语言还要难！

本书使用 Python 语言提供的标准 GUI 工具包：Tkinter 模块①。这个模块的名称来历是这 样的：原先有一种流行的跨平台 GUI 工具包 Tk，现在 Tkinter 模块通过定义一些类和函数， 实现了一个在 Python 中使用 Tk 的编程接口。可以简单地说，Tkinter 就是 Python 版的 Tk。

GUI 编程一般需要如下几个步骤：

1. 设计界面外观：这包括创建窗口和其他各种构件，并进行合适的布局。这一步与其 说是程序设计，不如说是美工设计。在流行的 Visual Basic、Eclipse 等集成开发环境 中，这一步只需用鼠标点击、拖放、调整大小就能完成。

2. 为每个构件定义事件处理程序：这一步是 GUI 开发的核心任务，决定着程序的功能 和与用户交互时的行为。

3. 编写应用程序的启动和总控部分：进行必要的初始化工作之后，进入主循环。 不同应用程序的用户界面虽然肯定会有不同，但构件的选择和布局是有很多共性的。读者如果用过一些 Windows 应用程序（如 MS Office 中的各种程序）的话，一定会发现众多 Windows 程序在界面风格方面的雷同。以下我们虽然用 Tkinter 来实现 GUI，但各种构件的 用法和布局的讨论是有普遍意义的。

GUI 工具包一般都利用面向对象技术实现的，即构件都是对象，具有属性和方法。构件 对象的属性用来记录构件的各种数据特性，构件对象的方法实现构件的行为特性。

8.2.2 初识 Tkinter

Tkinter 模块中定义了许多构件类，利用这些构件类可以容易地创建构件实例，从而建立 图形界面。下面按字母顺序列出一些常用的构件类及其简要功能：

- Button：按钮。用于执行命令。
- Canvas：画布。用于绘图、定制构件。
- Checkbutton：勾选钮。用于表示是否选择某个选项。
- Entry：录入框。用于输入、编辑一行文本。
- Frame：框架。是容器构件，用于构件组合与界面布局。
- Label：标签。用于显示说明文字。
- Listbox：列表框。用于显示若干选项。
- Menu：菜单。用于创建下拉式菜单或弹出式菜单。
- Message：消息。类似标签，但可显示多行文本。
- Radiobutton：单选钮。用于从多个冲突选项中选择一个。
- Scrollbar：滚动条。用于滚动显示更多内容。
- Text：文本区。用于显示和编辑多行文本，支持嵌入图像。
- Toplevel：顶层窗口。是容器构件，用于多窗口应用程序。

> ① 其他常用的 GUI 工具包有 wxPython，PyQt 等。

为了使用 Tkinter 中定义的构件类，需要先导入 Tkinter 模块。下面两种形式的 import 语 句都可以，一般来说后一种形式更方便。

```
import Tkinter
from Tkinter import *
```

首先看一个最简单的 Tkinter 程序：

【程序 8.1】eg8_1.py

```
from Tkinter import *
root = Tk()
root.mainloop()
```

程序 8.1 的第 1 行导入 Tkinter 模块。第 2 行调用 Tk 构造器，初始化 Tk 并自动创建一个 根窗口。根窗口的形状依赖于操作系统平台，一般都具有标题栏和最小化、最大化、关闭按钮。每个程序必须有也只有一个根窗口，并且要先于其他构件创建①，其他构件都是根窗口的子构件。第 3 行进入主循环，准备处理事件。除非用户关闭窗口，否则程序将一直处于主循环中。注意：只有进入了主循环，根窗口才可见②。程序的执行结果如图 8.3 所示：

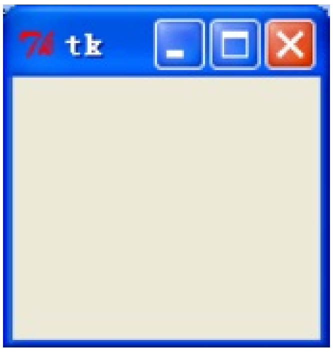


图 8.3 程序 8.1 的执行结果

程序 8.1 只搭建了一个图形界面雏形，除了根窗口，界面中没有任何能与用户进行交互 的构件。下面这个程序略有改进，在窗口中添加了一个标签构件：

【程序 8.2】eg8_2.py

```
from Tkinter import *
root = Tk()
aLabel = Label(root, text="Hello World")
aLabel.pack()
root.mainloop()
```

程序 8.2 的第 3 行创建标签构件 aLabel，Label 构造器的第一个参数 root 表示该标签构件 是根窗口的子构件，第二个参数指定标签的文本内容。第 4 行表示用 Pack 布局管理器对标签构件进行布局，这使得标签在根窗口中以紧凑的方式摆放。程序的执行效果如图 8.4 所示：



图 8.4 程序 8.2 的执行结果

大多数构件在创建之后并不会立即显示在窗口中，必须经由布局管理器进行布置之后才变成可见的，因此多数构件都要像上例中的标签构件一样经历创建和布局两个步骤。

> ① 事实上，如果程序没有显式创建根窗口而直接去创建其他构件，系统仍然会自动创建根窗口。
>
> ② 如果在命令行环境中交互式执行程序语句，窗口和其他构件无需进入主循环就能显示。

构件对象属性

每个构件都是对象，构造对象时设置的各种参数都是对象的属性（实例变量），如上例中 标签构件的 text 属性。Tkinter 构件一般都有许多属性，在用构造器创建构件实例时可以为一 些属性设置属性值，而没有设置的属性也都有预定义的缺省值。由于属性众多，又是有选择 地设置，所以创建实例时适合采用“命名参数”方式来传递属性值，即“属性=属性值”的 形式①。属性值如果是数值或单个字符，可以不用引号；如果是数值与字母混用，或者是字 符串，则必须用引号。

构件对象的属性值既可以在创建时指定，也可以在将来任何时候设置或修改。每种构件 类都提供 configure（或简写为 config）方法用于修改属性值。例如，如果在程序 8.2 中倒数 第 2 行处增加一条语句：

```
aLabel.config(text="Goodbye")
```

则执行程序后就会看到标签文本从“Hello World”变成了“Goodbye”。

Tkinter 还提供了另一种修改构件对象属性值的方法：将对象视为一个字典②，该字典以 属性名作为键，以属性值作为键值。按照修改字典键值的语法，上面这条语句可以写成：

```
aLabel["text"] = "Goodby"
```

用字典方法每次只能修改一个属性的值，而用 config 每次可以修改多个属性的值，例如 下面这条语句同时修改了标签的文本、前景色和背景色：

```
aLabel.config(text="Goodbye",fg="red",bg="blue")
```

根窗口的标题和大小

从图 8.3 和图 8.4 可见，根窗口默认的窗口标题是 Tk，如果对此不满意，可以通过调用 根窗口对象的 title 方法来设置窗口标题：

```
root.title("My GUI")
```

根窗口的默认大小是宽度和高度各为 200 像素，如果对此不满意，可以通过调用根窗口 对象的 geometry 方法来设置窗口大小。指定窗口尺寸的最简单形式是“宽度 x 高度”：

```
root.geometry("400x400")
```

父构件与子构件

如前所述，图形界面中的所有构件按父子关系构成了树状层次，构件之间的父子信息记录在每个构件都有的 master 和 children 属性中，Tkinter 会自动维护这两个属性的值。编程时，可以利用这两个属性来从子构件找到其父构件，从而间接引用父构件。例如语句

```
aLabel.master.title("My GUI")
```

的意思是对标签构件 aLabel 的父构件（即 root）调用 title 方法来设置窗口标题。

图形界面中当然不会只有一个构件，仿照上面创建标签构件的过程，我们可以根据需要 创建多个构件并在窗口中布局，最终设计出复杂的图形界面。例如：

【程序 8.3】eg8_3.py

```
from Tkinter import *
root = Tk()
aLabel = Label(root,text="Hello World")
aLabel.pack()
aButton = Button(root,text="Click Me")
aButton.pack()
root.mainloop()
```

> ① 参见第 4 章关于函数参数传递的内容。
>
> ② 回顾第 6 章关于字典类型的介绍：字典就是“键－值”配对的集合。

程序 8.3 的第 3－6 行分别创建了一个标签和一个按钮构件，并在窗口中用 Pack 布局管 理器进行布局。执行结果如图 8.5 所示：



图 8.5 程序 8.3 的执行结果 通过以上几个简单例子可以看出，基于 Tkinter 的图形界面设计非常简单，可概括为创建

构件并在窗口中进行布局的过程。接下来我们就要具体学习各种图形构件的使用方法和布局方法。

8.2.3 常见 GUI 构件的用法

本节介绍一些最常见的 GUI 构件的用法。为了便于讨论，我们使用 Python 的命令行界 面①来交互式地执行语句，这样可以执行一条语句就立即看到其执行效果。读者也可以一边 阅读本书，一边在计算机上动手练习。

GUI 编程首先要做的是导入 Tkinter 模块并创建根窗口：

```
from Tkinter import *
root = Tk()
```

可以看到，屏幕上立即出现一个根窗口，接下去就可以向根窗口中加入子构件了。

当所有构件添加完毕，通过执行下列语句来启动主循环：

```
root.mainloop()
```

这时可以看到 Python 命令行界面的>>>提示符没有了，表明现在 Tkinter 程序接管了控制权。 当最后关闭根窗口时，所有构件也都同时撤销。 下面演示各种构件的用法时，我们既可以在同一个根窗口中演示多个构件的用法，也可以每次演示新构件时重新创建根窗口。由于这两种做法并不影响本节的主旨，为了行文简洁， 我们对此不加区别，以上这几个导入模块、创建根窗口和进入主循环的步骤也不重复列出， 读者练习时可根据需要补上有关步骤。

标签

标签用于在窗口中显示文本信息，Tkinter 定义了类 Label 来支持标签构件的创建。创建 标签时需要指定父构件和文本内容，前者由 Label 构造器的第一个参数指定，后者用属性 text 指定。例如前面已经见过的：

```
>>> aLabel = Label(root,text="Hello World")
```

这条语句创建了一个标签构件实例，但该构件在窗口中仍然不可见。为使构件在窗口中 可见，需要调用布局管理器对该构件进行位置安排。仍然如前面已经见过的，下面这条语句 对标签对象 aLabel 调用方法 pack，意为用 Pack 布局管理器来安排这个标签的位置：

> ① Python 的 GUI 环境 IDLE 本身是用 Tkinter 写的程序，因此无法在 IDLE 中交互式地创建 Tkinter 窗口和其 他构件。

```
>>> aLabel.pack()
```

于是标签在根窗口中得到显示，同时根窗口的大小也改变了——变成刚好可以放置新加入的标签构件（参见图 8.4）。这正是 Pack（“打包”）的效果，即所有东西以紧凑的方式布置。Pack 布局管理器简单易用，但不适合进行复杂布局，我们后面会介绍其他布局管理器。

标签构件除了 text 属性之外，还有其他许多属性。上例中只为标签的 text 属性提供了值“Hello World”，其他属性（如字体、颜色等）都使用缺省值。下面这条语句为标签的更多属性设置属性值：

```
>>> Label(root,text='Text in Red',fg='red',width=40).pack()
```

其中属性 fg 和 width 分别表示标签文本的颜色和标签的宽度，效果见图 8.6：



图 8.6 标签构件

注意，上面这条语句采用了将对象创建和对象方法调用结合在一起的语法①：先用构造器 Label(…)创建一个标签对象，然后直接对这个对象调用方法 pack()，而不是先定义一个变 量作为新建对象的引用，然后通过变量引用来调用对象方法。显然，对象创建与对象方法调 用合并的写法不适合需要多次引用一个对象的场合。另外，初学者容易犯的一个错误是写成：

```
w = Label(...).pack()
```

这条语句实际上是将 pack 的返回结果（None）赋值给 w，而非 Label(…)的返回结果（新建的标签对象）。为了避免这类微妙的错误，最好将创建构件和构件布局的语句分开。

如果希望在程序运行过程中改变标签的文本内容，可以利用 8.2.2 中介绍的设置构件对象 属性的方法来重新设置标签对象的 text 属性值。很多 GUI 应用程序的窗口底部都有一个状态 栏（例如 Word 窗口的状态栏），用来显示程序的一些状态信息，这可以用标签构件来实现，状态的变化可以通过修改标签对象的 text 属性值来实现。

按钮

按钮也叫命令按钮，是界面中的一个矩形区域（通常长度大于高度），矩形内部是描述性 的标题(例如常见的“确认”和“取消”)。对按钮的操作是用鼠标点击，其作用是命令程序 去执行预定义的操作。按钮可以说是图形界面中最常见的构件，是用户命令程序执行某项任 务的基本手段。

下列语句在根窗口 root 中创建了一个按钮构件：

```
>>> quitButton = Button(root,text="Quit",command=root.quit)
```

对按钮构件来说是最重要的属性是 command，它用于将按钮与某个函数或方法进行关 联。传递给 command 的值是一个函数（或方法），该函数就是点击按钮时要执行的操作。上 例中将按钮与根窗口 root 的内建方法 quit 相关联，其功能是退出主循环。注意，传递给 command 属性的是函数对象，函数名后不能加括号，切记 f 是函数对象，而 f()是函数调用（的 结果）。

按钮构件在窗口中的位置也需要用布局管理器来安排，例如用 pack 布局管理器：

```
>>> quitButton.pack()
```

从显示结果（图 8.7）可以看到，与前述标签的情形一样，quitButton 按钮在窗口中以紧凑方式布置并变得可见。由于 pack 方法是在 Tkinter 的基类中定义的，而所有构件都是这个基类的子类，从而都继承了这个方法，所以对标签和按钮以及其他各种构件都可以调用 pack 方法。

> ① 参见第 7 章。



图 8.7 按钮构件

为了验证按钮的功能，我们先进入主循环：

```
>>> root.mainloop()
```

然后点击 Quit 按钮，可以看到又回到了 Python 解释器的提示符状态，这就是 root.quit 方法的作用。

实际应用程序中通常由程序员自己定义与按钮相关联的函数，以实现某种操作。下面这 个例子为按钮定义了一个简单的显示信息的函数：

```
>>> def hiButton():
...     print 'hi there'
...
>>> Button(root,text='print',command=hiButton).pack()
```

如果点击按钮，会看到在控制台显示信息“hi there”。 按钮构件还有其他许多属性，如宽度、文本颜色、按钮边框的 3D 风格、活动状态等等，

更多细节可参看本章后面的附录。

勾选钮

勾选钮也称为勾选框或复选框，用于向用户提供一个选项，用户对该选项有“选中”或 “不选”两种选择。勾选钮在外观上由一个小方框和一个相邻的描述性标题组成，未选中时 方框为空白，选中时方框中出现勾号，再次选择一个已打勾的勾选钮将取消选择。对勾选钮 的选择操作一般是用鼠标点击小方框或标题。Tkinter 的类 Checkbutton 实现了勾选钮构件， 其最简单的用法如下：

```
>>> Checkbutton(root,text="A Choice").pack()
```

如果程序中需要查询和设置选项的状态，可以将勾选钮与一个控制变量关联。控制变量 是 IntVar 类的实例，值为 1 或 0，分别对应选中和未选中状态。用法如下：

```
>>> v = IntVar()
>>> Checkbutton(root,text="A Choice",variable=v).pack()
```

程序中可以通过 v.get()和 v.set()来查询或设置勾选钮的状态。 在实际应用中，通常将多个勾选钮组合为一组，为用户提供多个相关的选项，用户可以选中 0 个、1 个或多个选项。例如：

```
>>> Checkbutton(root,text="Math").pack()
>>> Checkbutton(root,text="Python").pack()
>>> Checkbutton(root,text="English").pack()
```

执行效果如图 8.8 所示。为了在程序中获得各勾选钮的状态，可以为每个勾选钮关联一个不同的控制变量。

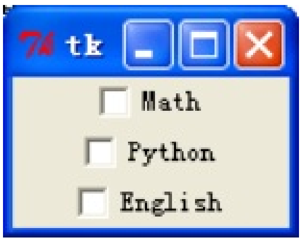


图 8.8 勾选钮构件

图 8.8 中各勾选钮排列的不太美观，这是因为 pack 在布局时默认采用了居中对齐方式。 布局管理器提供了其他对齐方式，详见 8.2.4。此外，在实际 GUI 设计中，为了表示一组勾 选钮的相关性，通常会用一个框架容器将它们组合起来（参见图 8.11）。

单选钮

和勾选钮类似，单选钮也是列出选项供用户选择，并且通常也是由若干个单选钮构成一 组来提供多个相关的选项。但与勾选钮不同的是，同组的单选钮在任意时刻只能有一个被选 中；每当换选其他单选钮时，原先选中的单选钮即被取消①。

单选钮的外观是一个小圆框加上相邻的描述性标题，未选中时圆框内是空白，选中时圆 框中出现一个圆点。对单选钮的选择操作是用鼠标点击小圆框或标题。Tkinter 提供的 Radiobutton 类可支持单选钮的创建。例如最简单的单选钮可用如下语句创建：

```
>>> Radiobutton(root,text="One").pack()
```

如上所述，实际应用中都是将若干个相关的单选钮组合成一个组，使得每次只能有一个 单选钮被选中。为了实现单选钮的组合，可以先创建一个 IntVar 或 StringVar 类型的控制变量，然后将同组的每个单选钮的 variable 属性都设置成该控制变量。由于多个单选钮共享一个控制变量，而控制变量每次只能取一个值，所以选中一个单选钮就会导致取消另一个。

为了在程序中获取当前被选中的单选钮的信息，可以为同组中的每个单选钮设置 value 属性的值。这样，当选中一个单选钮时，控制变量即被设置为它的 value 值，程序中即可通 过控制变量的当前值来判断是哪个单选钮被选中了。还要注意，value 属性的值应当与控制变 量的类型匹配：如果控制变量是 IntVar 类型，则应为每个单选钮赋予不同的整数值；如果控 制变量是 StringVar，则应为每个单选钮赋予不同的字符串值。下面这个例子将三个单选钮组 合成一组：

```
>>> v = IntVar()
>>> v.set(1)
>>> Radiobutton(root,text="One",variable=v,value=1).pack()
>>> Radiobutton(root,text="Two",variable=v,value=2).pack()
>>> Radiobutton(root,text="Three",variable=v,value=3).pack()
```

其中三个单选钮共享一个 IntVar 型的控制变量 v，且分别设置了值 1、2、3。第二行 v.set(1) 的作用是设置初始默认选项。上述语句序列的执行效果如图 8.9 所示。另外，图中三个单选钮是居中对齐，不太美观，可以用布局管理器设置对齐方式，详见 8.2.4。



图 8.9 单选钮构件

文本编辑区

文本编辑区是允许用户输入和编辑数据的区域，用户使用键盘在这个区域中输入文本，

输入过程中随时可以进行编辑，如光标定位、修改、插入等。有的文本编辑区只允许输入一行文本，有的则允许输入多行。文本编辑区一般可以设置行和列的大小，并且可以通过滚动条来显示、编辑更多的文本。

> ① 单选钮的英文是 radio button，名称源自收音机中预置电台按钮，选一个台的同时就取消了另一个台。

Tkinter 提供的 Entry 类可以实现单行文本的输入和编辑，不妨称之为录入框。下面的语 句创建并布置一个录入框构件：

```
>>> Entry(root).pack()
```

运行结果是在窗口中出现一行空白区域，点击此区域后会出现一个闪烁的光标，这时就可以在其中输入文本了。图 8.10 是输入了一些文本后的效果：



图 8.10 录入框构件

当用户输入了数据之后，应用程序显然需要某种手段来获取用户的输入，以便对数据进行处理。为此，可以通过 Entry 对象的 textvariable 属性将录入框与一个 StringVar 类型的控制 变量相关联，具体做法如下：

```
>>> v = StringVar()
>>> e = Entry(root,textvariable = v)
>>> e.pack()
```

此后程序中就可以利用 v.get()来获取录入框中的文本内容。假设用户在录入框内键入了文本“hello”，那么就有

```
>>> print v.get() hello
```

另外，程序中还可以通过 v.set()来设置录入框的内容：

```
>>> v.set('new text')
```

可以看到录入框中的文本立即变成了“new text”。 很多应用程序利用录入框作为用户登录系统时输入用户名和密码的界面元素，其中密码录入框一般不回显用户的输入，而是用“”代替，这在 *Tkinter* 中很容易做到，只需将录入 框对象的 *show* 属性设置为“”即可：

```
e.config(show='*')
```

除了 Entry 类，Tkinter 还提供一个支持多行文本录入与编辑的文本区构件类：Text。两 者的用法是类似的：

```
>>> Text(root).pack()
```

运行结果是在窗口中出现了一个多行的空白区域，在此区域可输入、编辑多行文本。Text 构件的用途非常多，用法也比 Entry 复杂，这里就不详细介绍了。

框架

利用前面几节介绍的基本构件虽然能够实现简单的图形界面，但不足以搭建出美观的复 杂图形界面。为了将基本构件在界面中有层次地组织起来，还需要构件容器。如我们在 8.1.2 中描述的，框架就是一种容器，其主要用途是将一组相关的基本构件组合成为一个“复合” 构件。利用框架对窗口进行模块化分隔，即可建立复杂的图形界面结构。每个框架都是一个 独立的区域，可以独立地对其中包含的子构件进行布局。

Tkinter 提供了 Frame 类来创建框架构件，框架的宽度和高度分别用 width 和 height 属性 来设置，框架的边框粗细用 bd（或 border、borderwidth）属性来设置（默认值为 0，即没有 边框），边框的 3D 风格用 relief 属性来设置（默认是与环境融合的 flat 风格，其他可选的值 还有 groove、sunken、raised 和 ridge）。框架构件和基本构件一样需要先创建再布局，例如：

```
>>> f = Frame(root,width=300,height=400,bd=4,relief="groove")
>>> f.pack()
```

这个框架的边框风格是 groove，读者可以试试其他边框风格分别是什么样子的。顺便说一下， relief 属性也适用于按钮构件。

下面这个例子演示了如何将框架用作容器来组合构件，图 8.11 是其执行效果。

```
>>> f = Frame(root,bd=4,relief="groove")
>>> f.pack()
>>> Checkbutton(f,text="Math").pack()
>>> Checkbutton(f,text="Python").pack()
```

```
>>> Checkbutton(f,text="English").pack()
```

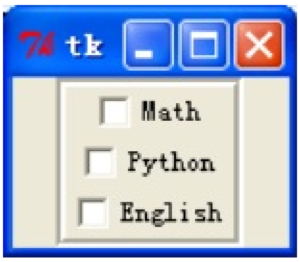


图 8.11 利用框架组合构件

除了作为容器来组合多个构件的用途，框架还可用于图形界面的空间分隔或填充，例如：

```
>>> Label(root,text="one").pack()
>>> Frame(height=2,bd=1,relief="sunken").pack(fill=X,pady=5)
```

```
>>> Label(text="two").pack()
```

其中第二行语句定义的框架只起着分隔两个标签构件的作用，该框架在根窗口中以 x 方向填满（fill=X）、y 方向上下各填充 5 个像素空间（pady=5）的方式进行 Pack 布局。效果见图 8.12：



图 8.12 利用框架分隔构件

后文（见 8.4.1）会介绍，我们经常将 GUI 应用程序封装成类，尤其是封装成 Frame 的 子类。这样做的好处是可以将程序逻辑与用户界面融为一体，界面元素之间的交互情况对外 部是隐藏的。

菜单

菜单也是 GUI 最常用的构件之一。就像饭店里用的菜单一样，菜单构件是一个由许多菜 单项组成的列表，每个菜单项表示一条命令或一个选项。用户通过鼠标或键盘选择菜单项， 以执行命令或选中选项，由此可见菜单兼具命令按钮、勾选钮和单选钮的功能。另一方面， 菜单又不像大量命令按钮、勾选钮和单选钮那么占空间，因为菜单在不用时通常是“合上” 的，在界面中只是一个占用极少空间的菜单按钮，直到用户点击菜单按钮才会展开整个菜单。 图形界面中一般有多个菜单，它们通常以相邻的方式布置在一起，形成窗口的菜单栏，并且 一般置于窗口顶端。

除了菜单栏里的菜单，GUI 中还常用一种弹出式菜单，这种菜单平时在界面中是不可见 的，当用户在界面中点击鼠标右键时才会“弹出”一个与点击位置相关的菜单。

有时候菜单中一个菜单项的作用是展开另一个菜单，形成“级联式”菜单。

Tkinter 提供 Menu 类用于创建菜单构件，具体用法是先创建一个菜单构件实例，并与某 个窗口（根窗口或者顶层窗口）进行关联，然后再为该菜单添加菜单项。与根窗口关联的菜 单实际上构成了根窗口的菜单栏。菜单项可以是简单命令、级联式菜单、勾选钮或一组单选 钮，分别用 add_command、add_cascade、add_checkbutton 和 add_radiobutton 来添加。为了使 菜单结构清晰，还可以用 add_separator 在菜单中添加分割线。

例如，下列语句以交互方式创建了一个菜单（假设已经创建了根窗口 root）：

```
>>> m = Menu(root)
>>> root.config(menu=m)
>>> m.add_command(label="File")
>>> m.add_command(label="Edit")
```

其中第一条语句创建菜单实例 m（作为 root 的子构件）；第二条语句将 root 的 menu 属性设置 为 m ①，这导致将 m 布置于根窗口的顶部，形成菜单栏；第三、四条语句创建 m 的两个命 令菜单项。执行结果如图 8.13 所示。另外从上面的语句可以看到，菜单构件与前面介绍的构 件都不同，不需要调用布局管理器来使之可见，Tkinter 会自动布局并显示菜单。



图 8.13 菜单构件

上面这个例子其实没什么用，因为没有为命令菜单项指定相应的处理代码。另外，在实际应用中，窗口菜单栏里的菜单项通常是一个级联菜单，而不是简单命令菜单项。下面看一 个更实用的例子：

【程序 8.4】eg8_4.py

```
from Tkinter import *
def callback():
    print "hello from menu"
root = Tk()
m = Menu(root)
root.config(menu = m)
filemenu = Menu(m)
m.add_cascade(label="File", menu=filemenu)
filemenu.add_command(label="New", command=callback)
filemenu.add_command(label="Open...", command=callback)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=callback)
helpmenu = Menu(m)
m.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About...", command=callback)
mainloop()
```

> ① 也可采用 root['menu'] = m 的语法，参见 8.2.3.1。

程序 8.4 首先以根窗口为父构件创建菜单构件 m，接着将 m 设置为根窗口的菜单栏；然 后以 m 为父构件创建另两个菜单构件 filemenu 和 helpmenu，它们分别构成菜单 m 的菜单项 “File”和“Help”的级联菜单。菜单 filemenu 又由三个命令菜单项组成（中间有一道分隔 线），菜单 helpmenu 中只有一个命令菜单项。各个菜单在界面中的位置由 Tkinter 自动布局， 不需

要调用布局管理器。为了简化程序，我们将所有命令菜单项都关联到同一个函数 callback，实际应用程序当然应该为每个命令编制各自的处理函数。执行本程序并点击 File 菜单项之后的结果如图 8.14 所示：



图 8.14 程序 8.4 的执行结果

顶层窗口

迄今我们所写的程序都只有一个窗口，即根窗口。如所熟知的，像 Word 之类的应用程 序是多窗口的，每打开一个文档都新开一个窗口。为了支持多窗口应用程序，Tkinter 提供 Toplevel 类用于创建顶层窗口构件。顶层窗口的外观与根窗口一样，可以独立地移动和改变大小，并且不需要像其他构件那样必须在根窗口中进行布局后才显示。一个应用程序只能有一个根窗口，但可以创建任意多个顶层窗口。例如：

```
>>> root = Tk()
>>> Label(root,text="hello").pack()
>>> top = Toplevel()
>>> Label(top,text="world").pack()
```

这个语句序列先创建根窗口，在根窗口中创建一个标签，然后创建了一个顶层窗口，又在顶层窗口中创建了一个标签。执行结果如图 8.15 所示。

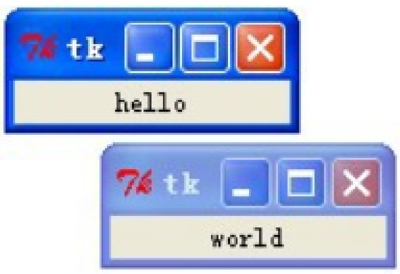


图 8.15 顶层窗口

对上例要说明的是，虽然创建 Toplevel 构件 top 时没有指定以根窗口 root 作为父构件，但 top 确实是 root 的子构件，因此关闭 top 并不会结束程序，因为根窗口仍在工作；但若关闭根窗口，则包含 top 在内的整个界面都会关闭。所以顶层窗口虽然具有相对的独立性，但它不能脱离根窗口而存在。即使在没有根窗口的情况下直接创建顶层窗口，系统也会自动先创建根窗口。

与根窗口类似，可以调用 Toplevel 类的 title 和 geometry 方法来设置它的标题和大小：

```
>>> top.title('hello toplevel')
>>> top.geometry('400x300')
```

当然也可以为顶层窗口建立菜单，方法和根窗口类似，这里就不赘述了。

8.2.4 布局

布局指的是界面元素在界面中的位置安排。Tkinter 中提供了布局管理器，其任务是根据 程序员的要求以及其他一些约束来安排构件的位置。使用布局管理器的优点是程序员不需要 了解底层显示系统的细节，可以在较高层次上考虑界面布局问题。

如前所述，多数构件在创建之后还需进行布局才会显示在屏幕上，即要经过两个步骤：

```
w = Constructor(parent,...) w.GeometryManager(...)
```

其中 Constructor 是构件类（如 Label，Button 等），parent 是父构件，GeometryManager 是布局管理器方法。Tkinter 提供了三种布局管理器：Pack、Grid 和 Place。

Pack 布局管理器

Pack 布局管理器以紧凑的方式将构件在窗口中“打包”，调用构件的 pack 方法即可以这 种方式布局。具体的打包方式可以用一个比方来说明：设想窗口是由弹性材料制成的，当要 放入一个构件时，就先把窗口空间撑大到足够容纳该构件，然后将构件紧贴内部的某条边（缺 省是顶边）放入，然后重复此过程不断放入构件。可见，在缺省情形下，放入同一个窗口的 所有构件是沿垂直方向自顶向下一个紧贴一个进行布置的，但可以通过 pack 方法的 side 选项设置成沿水平方向打包。

例如，执行下列语句后得到的布局效果如图 8.16(a)所示。

```
>>> Label(root,text="Input a number:").pack()
>>> Entry(root).pack()
>>> Button(root,text="OK").pack()
```

而执行下列语句后的布局效果如图 8.16(b)所示，这里用到了 side 属性：side="left"使得构件贴着左边放置。

```
>>> Label(root,text="Input a number:").pack(side="left")
>>> Entry(root).pack(side="left")
>>> Button(root,text="OK").pack(side="left")
```

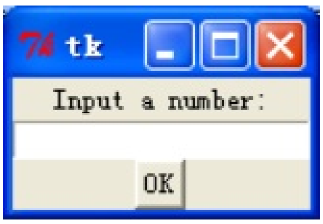




图 8.16 Pack 布局管理器

如果对窗口中的所有基本构件以打包方式布局，将使大大小小的构件形成一摞，显然不 美观。实际应用中常见的做法是对框架进行打包布局，一个一个框架像集装箱一样并排或堆叠，然后再将基本构件作为各框架的子构件进行布局，这样能使界面整齐美观。

虽然 Pack 管理器还提供其他布局选项，但在此不多介绍了，因为我们有更灵活、更好用 的布局管理器：Grid 和 Place。

pack 方法的“逆”方法是 pack_forget 方法，意为将用 pack 布局的构件从界面中拿掉， 从而构件变成不可见。注意，这时构件作为对象仍然存在，只是未显示在界面中而已，我们 随时可以再次调用任何布局管理器方法来使构件可见。

rid 布局管理器*

Grid 布局管理器将窗口或框架视为一个由行和列构成的二维表格，并将构件放入行列交 叉处的单元格中。为了使用这种布局，只需先创建构件，再用 grid 方法的选项 row 和 column 指定行、列编号即可。不需要预先定义表格的尺寸，Grid 布局管理器会根据构件的大小自动 调整：每一列的宽度由该列中最宽的构件决定，每一行的高度由该行最高的构件决定。行、列都是从 0 开始编号，row 的缺省值为当前下一空行，column 的缺省值总为 0。可 以在布置构件时指定不连续的行号或列号，这相对于预留了一些行列，但这些预留的行列是 不可见的，因为行列上没有构件存在，也就没有宽度和高度。

Grid 布局管理器的使用非常简单，可以说是进行图形界面布局的首选工具。先看一个简 单例子：

```
>>> Label(root,text="ID Number:").grid()
>>> Label(root,text="Name:").grid()
>>> Entry(root).grid(row=0,column=1)
>>> Entry(root).grid(row=1,column=1)
```

其中第一条语句使用缺省行号 0 和缺省列号 0 来安排标签“ID Number:”，第二条语句使用缺省行号 1 和缺省列号 0 来安排标签“Name”，后面两条语句在指定的位置安排录入框。布局效果如图 8.17 所示：

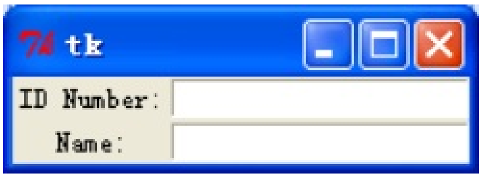


图 8.17 Grid 布局

从图 8.17 可以看出，标签构件在单元格里是居中放置的，如果觉得这种对齐方式不好看， 可以用 grid 方法的 sticky 选项来改变对齐方式。

Tkinter 中常利用“方位”概念来指定对齐方式，具体方位值包括 N、NE、E、SE、S、 SW、W、NW 和 CENTER（见图 8.18）。将 sticky 选项设置为某个方位，就表示将构件沿单元格的某条边或某个角对齐。

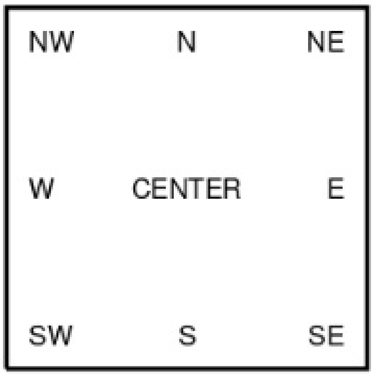


图 8.18 方位

如果构件比单元格小，未能填满单元格，则可以指定如何处理多余空间，比如在水平方 向或垂直方向拉伸构件以填满单元格。可以利用方位值的组合来使构件延伸，例如若将 sticky 设置为 E+W，则构件将在水平方向延伸，占满单元格的宽度；若设置为 E+W+N+S（或 NW+SE），则构件将在水平和垂直两个方向延伸，占满整个单元格。

如果想让一个构件占据多个单元格，可以使用 grid 方法的 rowspan 和 columnspan 选项来 指定在行和列方向上的跨度。例如，下面的语句序列能产生如图 8.19 所示的复杂布局：

```
>>> Label(root,text="ID Number:").grid(sticky=E)
>>> Label(root,text="Name:").grid(sticky=E)
>>> Entry(root).grid(row=0,column=1)
>>> Entry(root).grid(row=1,column=1)
>>> Checkbutton(root,text="Registered User").grid(
... columnspan=2,sticky=W)
>>> Label(root,text="X").grid(row=0,column=2,
... columnspan=2,rowspan=2,sticky=W+E+N+S)
>>> Button(root,text="Zoom In").grid(row=2,column=2)
>>> Button(root,text="Zoom Out").grid(row=2,column=3)
```



图 8.19 利用 Grid 进行复杂布局

下面再看一个利用框架来实现复杂界面结构的例子：

```
>>> f1 = Frame(root,width=100,height=100,bd=4,relief="groove")
>>> f1.grid(row=1,column=1,rowspan=2,sticky=N+S+W+E)
>>> Checkbutton(f1,text="PC").grid(row=1,sticky=W)
>>> Checkbutton(f1,text="Laptop").grid(row=2,sticky=W)
>>> f2 = Frame(root,width=100,height=50,bd=4,relief="groove")
>>> f2.grid(row=1,column=2,columnspan=2,sticky=N)
>>> b1 = Button(root,text="OK",width=6)
>>> b1.grid(row=2,column=2,sticky=E+W,padx=2)
>>> b2 = Button(root,text="Cancel",width=6)
>>> b2.grid(row=2,column=3,sticky=E+W,padx=2)
```

结果如图 8.20 所示。可以看出，这个例子大致实现了图 8.2 所要求的界面。



图 8.20 利用框架的布局

grid 方法的“逆”方法是 grid_forget 方法，意为将用 grid 布局的构件从界面中拿掉，从 而构件变成不可见。注意，这时构件作为对象仍然存在，只是未显示在界面中而已，我们随 时可以再次调用任何布局管理器方法来使构件可见。

Place 布局管理器*

Place 布局管理器直接指定构件在父构件中的位置坐标。为使用这种布局，只需先创建构件，再调用构件的 place 方法，该方法的选项 x 和 y 用于设定坐标。父构件（窗口或框架）的坐标系统以左上角为(0,0)，x 方向向右，y 方向向下。

由于(x,y)坐标确定的是一个点，而子构件可看作是一个矩形，这个矩形怎么放置在一个 点上呢？Place 通过“锚点”来处理这个问题：利用方位值（见图 8.18）指定子构件的锚点， 再利用 place 方法的 anchor 选项来将子构件的锚点定位于父窗口的指定坐标处。利用这种精 确的定位，可以实现一个或多个构件在窗口中的各种对齐方式。anchor 的缺省值为 NW，即 构件的左上角。例如下面两条语句分别将两个标签置于根窗口的(0,0)和(199,199)处，定位锚 点分别是（默认的）NW 和 SE：

```
>>> Label(root,text="Hello").place(x=0,y=0)
>>> Label(root,text="World").place(x=199,y=199,anchor=SE)
```

下列语句序列围绕根窗口的点(100,100)按不同锚点布置了若干个按钮：

```
>>> Button(root,text="CCCCCCCCCCCCCCCCCC").place(x=100,y=100,
... anchor=CENTER)
>>> Button(root,text=" NW ").place(x=100,y=100,anchor=NW)
>>> Button(root,text="E").place(x=100,y=100,anchor=E)
>>> Button(root,text="W").place(x=100,y=100,anchor=W)
>>> Button(root,text=" SE ").place(x=100,y=100,anchor=SE)
```

以上语句的执行结果如图 8.21 所示。

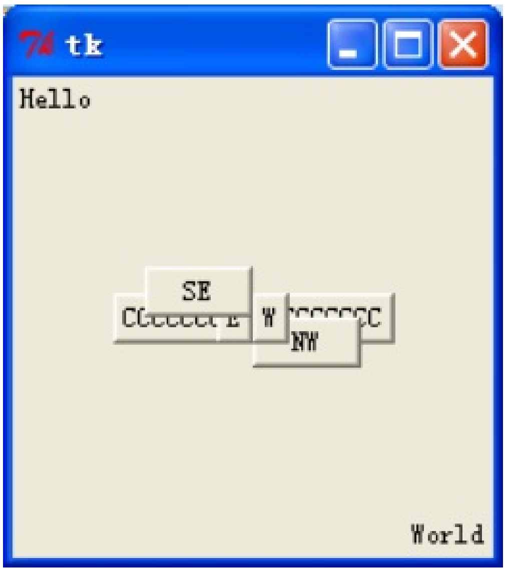


图 8.21 利用 Place 布局

Place 布局管理器既可以像上例这样用绝对坐标指定位置，也可以用相对坐标指定位置。 相对坐标通过选项 relx 和 rely 来设置，取值范围为 0～1，表示构件在父构件中的相对比例 位置，如 relx=0.5 即表示父构件 x 方向上的二分之一处。相对坐标的好处是当窗口改变大小时， 构件位置将随之调整，不像绝对坐标固定不变。例如下面这条语句将标签布置于水平方向四分之一、垂直方向二分之一处，定位锚点是 SW：

```
Label(root,text="Hello").place(relx=0.25,rely=0.5,anchor=SW)
```

除了指定构件位置，Place 布局管理器还可以指定构件大小。既可以通过选项 width 和 heightPlace 来定义构件的绝对尺寸，也可以通过选项 relwidth 和 relheight 来定义构件的相对尺寸（即相对于父构件两个方向上的比例值）。

Place 是最灵活的布局管理器，但用起来比较麻烦，通常不适合对普通窗口和对话框进行 布局，其主要用途是实现复合构件的定制布局。

place 方法的“逆”方法是 place_forget 方法，意为将用 place 布局的构件从界面中拿掉， 从而构件变成不可见。注意，这时构件作为对象仍然存在，只是未显示在界面中而已，我们 随时可以再次调用任何布局管理器方法来使构件可见。

8.2.5 对话框*

除了利用窗口中的各种构件之外，应用程序与用户进行交互的另一个重要手段是对话框。 对话框是一个独立的顶层窗口，通常是在程序执行过程中根据需要而“弹出”的窗口，用于 从用户获取输入或者向用户显示消息。

对话框分为两种类型：模态（modal）和非模态（modeless）对话框①。模态对话框在关 闭之前将阻止程序其他窗口的操作，而非模态对话框则不会阻止程序其他窗口的操作。模态 对话框常用于向用户警告重要信息，或者等待用户输入必需的数据（如登录用户名和密码、 打开或保存文件输入文件名等）。

Tkinter 提供若干标准模块用于创建弹出式对话框：

- tkMessageBox 模块：提供一系列用于显示信息或进行简单对话的消息框，可通过调 用函数 askokcancel、askquestion、askretrycancel、askyesno、showerror、showinfo 和 showwarning 来创建。
- tkFileDialog 模块：提供用于文件浏览、打开和保存的对话框，可通过调用函数 askopenfilename 和 asksaveasfilename 来创建。
- tkColorChooser 模块：提供用于选择颜色的对话框，可通过函数 askcolor 来创建。

> ① 这里“模态”的意思是：对话框影响着程序的某种执行模式（mode）、状态，因此必须完成对话才能继续。 非模态对话框则不表示特定程序模式，主程序可以不管对话结果而继续执行。

以上各种标准对话框模块的更多细节请参考本章附录或其他参考资料，在此我们只用几 个简单例子演示一下其用法。下面这个语句序列弹出一个简单的对话框（图 8.22①），然后根 据用户的回答进行适当处理。

```
>>> from Tkinter import *
>>> from tkMessageBox import *
>>> root = Tk()
>>> answer = askokcancel("Dialog","Is that OK?")
>>> if answer:
... print "OK"
```



图 8.22 askokcancel 对话框



下面这个语句尝试打开一个文件，如果打开失败则弹出一个报错消息框（图 8.23），并且还会发出报警声响：

```
>>> try:
...     f = open(filename)
... except:
...     showwarning("Open file","Open failed")
```

图 8.23 showwarning 消息框

标准对话框模块对于应用程序的对话框设计通常已经够用，但有些应用程序可能需要更复杂的对话框，这时我们可以定制对话框。设计自定义的对话框窗口与创建其他窗口并无本质上的不同，主要步骤都是先创建 Toplevel 构件，然后添加所需的输入区、按钮和其他构件。下面是一个定制消息框的简单程序：

> ① 从图中可见按钮文本是中文，这是因为 GUI 工具包都建立在底层操作系统的窗口管理器之上，本书作者的计算机使用中文 Windows，其底层窗口管理器为常见按钮自动提供了中文文本。如果是英文版操作系统，应该显示如函数名所提示的“OK”、“Cancel”、“Yes”、“No”等。

【程序 8.5】eg8_5.py

```
from Tkinter import *
def myMessageBox():
    top = Toplevel(height=200,width=400)
Label(top,text='hello').pack()
root = Tk()
Button(root,text='click me',command = myMessageBox).pack()
root.mainloop()
```

运行此程序，可看到根窗口中有一个“ click me ”按钮，点击该按钮将调用函数 myMessageBox，此函数首先创建一个顶层窗口，然后在其中添加一个标签，这相当于定制了一个简陋的消息框。

8.3 Tkinter 事件驱动编程

在 8.2 节中我们学习了图形用户界面中的各种构件的用法，至此我们已经能够为应用程 序搭建用户界面的外观部分，用户界面的另一个重要部分是各界面元素所对应的程序功能。 GUI 应用程序与普通应用程序的一个不同之处就在于，实现程序功能的代码与图形界面元素 相关联，这导致了一种新的程序执行模式——事件驱动。8.1.3 中简单介绍了事件驱动编程的 基本概念，现在我们来详细介绍 Tkinter 的事件驱动编程。

8.3.1 事件和事件对象

事件是针对应用程序所发生的事情，并且需要应用程序对它做出响应或进行处理。Tkinter中定义了很多种事件，足以支持常见的 GUI 应用程序开发。

Tkinter 事件可以用特定形式的字符串来描述，称为事件模式。事件模式的一般形式是：

```
<modifier-type-detail>
```

其中类型符 type 指定事件类型，最常用的类型有分别表示鼠标事件和键盘事件的 Button 和 Key；修饰符 modifier 用于描述鼠标键或键盘的双击、组合等情况；细节符 detail 指定具体的鼠标键或键盘按键，如鼠标的左中右三个键分别用 1、2、3 表示，键盘按键用相应字符或按键名称表示。modifier 和 detail 是可选的，而且事件模式经常可以使用简化形式。例如

```
<Double-Button-1>
```

描述符中，修饰符是 Double，类型符是 Button，细节符是 1，综合起来描 述的事件就是双击鼠标左键。

常用的鼠标事件包括：

- <ButtonPress-1>：按下鼠标左键。可简写为<Button-1>甚至<1>①。类似地有<Button-2>（按下鼠标中键）和<Button-3>（按下鼠标右键）。
- <B1-Motion>：按下鼠标左键并移动鼠标。类似有<B2-Motion>和<B3-Motion>。
- <Double-Button-1>：双击鼠标左键。
- <Enter>：鼠标指针进入构件。
- <Leave>：鼠标指针离开构件。

> ① 从易理解和简明的标准看，<Button-1>形式最可取。

常用的键盘事件包括：

- <Key-a>：按下 a 键。可简写为 a（不用尖括号！）。可打印字符（字母、数字和标点符号）都可像字母 a 这样使用，但有两个例外：空格键对应的事件是<space>，小于 号键对应的事件是<less>。注意：1 是键盘事件，而<1>是鼠标事件。
- <Return>：按下回车键。非可打印字符都可像回车键这样用<键名>表示对应事件，例如<Tab>、<Shift_L>、<Control_R>、<Up>、<Down>、<F1>等等。
- <Key>：按下任意键。
- <Shift-Up>：同时按下 Shift 键和↑键。类似的还有 Alt 组合、Ctrl 组合。

每个事件都导致系统创建一个 Event 对象，并将该对象传递给事件处理函数。事件对象 具有若干描述事件的属性，常用的有：

- x 和 y：鼠标点击位置坐标（相对于构件左上角），单位是像素。
- x_root 和 y_root：鼠标点击位置坐标（相对于屏幕左上角），单位是像素。
- num：点击的鼠标键号，1、2、3 分别表示左、中、右键。
- char：如果按下 ASCII 字符键，此属性即是该字符；如果按下特殊键，此属性为空 串。
- keysym：如果按下普通 ASCII 字符键，此属性即是该字符；如果按下特殊键，此属 性设置为该键的名称（是个字符串）。
- keycode：所按键的编码。注意，此编码无法区分该键上的不同字符，即它不是键上 字符的编码。
- keysym_num：这是 keysym 的数值表示。对普通单字符键来说，就是 ASCII 码。 例如，按下任意键都可触发<Key>事件，在事件处理函数中可以根据传递来的事件对象的 char 属性来确定具体按下的是哪一个键。

8.3.2 事件处理

GUI 应用程序的核心是对各种交互事件的处理程序。应用程序一般在完成建立图形界面 等初始化工作后都会进入一个事件循环，等待事件发生并触发相应的事件处理程序。Tkinter 程序通过 mainloop 方法进入事件循环，而事件与相应事件处理程序之间是通过绑定建立关联 的。

最常见的绑定形式是针对构件实例的：

```
<构件实例>.bind(<事件描述符>,<事件处理程序>)
```

其语义是：若针对<构件实例>发生了与<事件描述符>相匹配的事件，则调用<事件处理程序>。 调用事件处理程序时，系统会传递一个 Event 类的对象作为实际参数，该对象描述了所发生 事件的详细信息。

事件处理程序一般都是用户自定义的函数。这种函数在应用程序中定义，但不由应用程 序调用，而是由系统调用，所以一般称为回调（callback）函数。

GUI 应用程序经常封装为类，在这种情况下，事件处理程序常常定义为应用程序类的方 法。我们将在 8.4.1 中通过例子详细介绍这种做法。

先看一个处理鼠标点击事件的例子：

【程序 8.6】eg8_6.py

```
from Tkinter import *
def callback(event):
    print "clicked at", event.x, event.y
root = Tk()
f = Frame(root, width=100, height=100)
f.bind("<Button-1>", callback)
f.pack()
root.mainloop()
```

本程序在根窗口中添加了一个框架构件，然后把框架构件与<Button-1>事件进行了绑定， 对应<Button-1>事件的回调函数是 callback，意思是每当在框架中点击鼠标左键时，都将触发 callback 执行。系统执行 callback 时，将一个描述事件的 Event 类对象作为参数传递给该函数， 该函数从事件对象参数中提取点击位置信息并在控制台输出类似“clicked at 44 63”的信息。

键盘事件与焦点

当图形界面中存在许多构件时，如果是用鼠标直接点击某个窗口或构件，程序自然就知

道要操作哪个构件。但如果是按一下键盘，应该由哪个构件做出响应呢？GUI 引入了“焦点”概念：图形界面中有唯一焦点，任何时刻只能有一个构件占有焦点，键盘事件总是发送到当前占有焦点的构件。焦点的位置可以通过构件的 focus_set()方法来设置，也可以用键盘上的

Tab 键来轮转。因此，键盘事件处理比鼠标事件处理多了一个设置焦点的步骤，如下例所示：

【程序 8.7】eg8_7.py

```
from Tkinter import *
def printInfo(event):
    print "pressed", event.char
root = Tk()
b = Button(root,text = 'Press any key')
b.bind('<Key>',printInfo)
b.focus_set()
b.pack()
root.mainloop()
```

本程序创建了一个按钮构件，该按钮与按任意键事件<Key>进行绑定，事件处理程序是 回调函数 printInfo。此程序的 b.focus_set()语句将按钮设为键盘焦点，从而按下任何键都会由 按钮响应，并触发 printInfo 函数来处理事件，处理过程是显示按下的键的字符。读者可以思 考一下：本例中绑定的是<Key>事件，运行时如果输入上档键（如@#$%^&之类）会出现什 么结果呢？

绑定到多个事件 一个构件可以响应多种事件，例如下面这个程序同时响应鼠标和键盘事件：

【程序 8.8】eg8_8.py

```
from Tkinter import *
def callback1(event):
    print "pressed", event.char
def callback2(event): f.focus_set()
    print "clicked at", event.x, event.y
root = Tk()
f = Frame(root, width=100, height=100)
f.bind("<Key>", callback1)
f.bind("<Button-1>", callback2)
f.pack()
root.mainloop()
```

此程序在根窗口中创建一个框架构件，并为框架构件同时绑定了任意键事件<Key>和鼠 标左键事件<Button-1>。运行此程序，先在框架中点击鼠标，从而触发 callback2 函数的执行，该函数又将框架设置为键盘焦点。此后，按下任何键都将触发 callback1 函数的执行，其功能是显示所按的字符。运行此程序后如果没有在框架中先点击鼠标，则框架未获得焦点，也就不会对键盘事件进行处理。

当构件绑定的多个事件之间具有“特殊与一般”的关系，总是调用最“近”的事件处理 程序。例如，如果将某构件与任意键事件<Key>绑定，相应事件处理程序是 h1，又与回车键 事件<Return>绑定，相应事件处理程序是 h2，那么当按下回车键时，处理此事件的将是 h2。

绑定层次

前面三个例子中都是针对某个构件实例进行事件绑定，称为“实例绑定”。实例绑定只对 该构件实例有效，对其他实例——即使是同类型的构件——是无效的。除了实例绑定，Tkinter 还提供了其他事件绑定方式。实际上，Tkinter 中共有不同层次的四种绑定方法：

- 实例绑定：绑定只对特定构件实例有效，用构件实例的 bind 方法实现。
- 类绑定：绑定针对构件类，故对该类的所有实例有效，可用任何构件实例的 bind_class 方法实现。例如，为使 Button 类的所有实例都以同样方式响应回车键事件，可执行：

  ```
  root.bind_class("Button","<Return>",callback)
  ```

- 窗口绑定：绑定对窗口（根窗口或顶层窗口）中的所有构件有效。用窗口的 bind 方 法实现，例如为使窗口中所有构件都以同样方式响应鼠标右键点击事件，可执行：

  ```
  root.bind('<Button-3>',callback)
  ```

- 应用程序绑定：绑定对应用程序中的所有构件都有效。用任一构件实例的 bind_all 方法实现。例如，很多应用程序在运行时可以随时按下 F1 键以使用户得到帮助信 息，这可以通过建立 F1 键的应用程序绑定来实现：

  ```
  root.bind_all('<F1>',printHelp)
  ```

下面这个例子演示了事件传递与绑定层次结合所带来的后果：

【程序 8.9】eg8_9.py

```
from Tkinter import *
def printInstance(event):
    print 'Instance:',event.keycode
def printToplevel(event):
    print 'Toplevel:',event.keycode
def printClass(event):
    print 'Class:',event.keycode
def printApp(event):
    print 'Application:',event.keycode
root = Tk()
b = Button(root,text = 'Press Return')
b.bind('&lt;Return&gt;',printInstance)
b.winfo_toplevel().bind('&lt;Return&gt;',printToplevel)
root.bind_class('Button','&lt;Return&gt;',printClass)
root.bind_all('&lt;Return&gt;',printApp)
b.pack()
b.focus_set()
root.mainloop()
```

本程序中定义了四个层次的事件绑定，运行此程序并按下回车键，将得到如图 8.24 所示 的输出。这是因为<Return>事件首先被拥有焦点的按钮实例 b 捕获，并执行 printInstance 函 数。此后，<Return>事件还将向 b 的各级上层传递，从而依次被 b 所属的 Button 类、b 所属 的顶层窗口 root、b 所属的应用程序这三个层次捕获，分别导致 printClass、printTopleve 和 printApp 三个函数的执行。

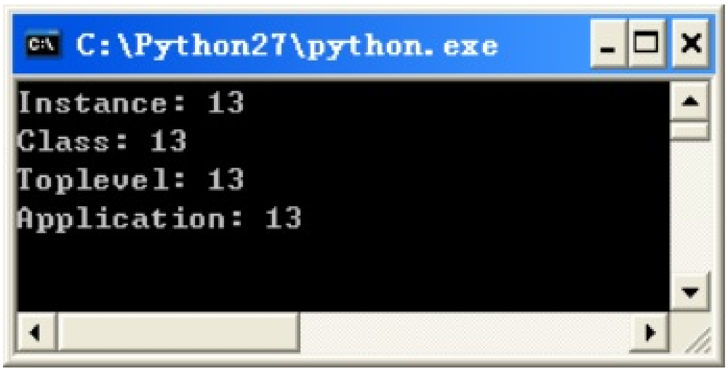


图 8.24 多层绑定

关于程序 8.9 还有几点要说明：（1）程序中的 b.winfo_toplevel()方法返回 b 所属的顶层构件，本例中即根窗口 root；（2）对程序代码与输出结果进行比较后可看出，事件的传递层次与程序中绑定语句的次序没有关系；（3）类绑定与应用程序绑定可以通过任何构件来设置，因此将上面程序中的 root.bind_class 和 root.bind_all 改成 b.bind_class 和 b.bind_all，结果也是 一样的。

协议处理

用过 Word 的读者都知道，如果编辑了文档还没有保存就去关闭程序窗口，Word 会弹出 一个对话框，询问用户是否要保存当前文档。如果我们希望利用事件绑定到事件处理程序来 实现这种功能，就面临一个问题："关闭窗口"并不属于前面介绍过的事件类型，因此无法用 事件绑定来处理。

为此，Tkinter 提供了一种称为"协议处理"的机制，用于应用程序处理来自操作系统窗 口管理器的协议消息。处理过程是这样的：当用户企图关闭窗口，操作系统的窗口管理器就 会生成一条 WM_DELETE_WINDOW 的协议消息并发送给应用程序，应用程序再调用相应的 处理程序来处理这条消息。

窗口构件有一个称为 protocol 的方法，用于定义对协议消息的处理程序：

```
<窗口构件>.protocol("WM_DELETE_WINDOW",<处理程序>)
```

其中窗口构件可以是根窗口或顶层窗口，处理程序是函数或方法。如此定义之后，当用户试图关闭窗口时，我们自己的处理程序就会接管控制。处理程序可以弹出一个消息框询问用户是否要保存当前数据，或者干脆忽略关闭窗口的请求。处理完毕之后，可以在处理程序中完成关闭窗口的操作，方法是调用窗口的 destroy 方法。例如：

【程序 8.10】eg8_10.py

```
from Tkinter import *
from tkMessageBox import *
def callback():
    if askokcancel("Quit","Do you really wish to quit?"):
        root.destroy()
root = Tk()
root.protocol("WM_DELETE_WINDOW", callback)
root.mainloop()
```

虚拟事件

我们也可以自定义新的事件类型，称为虚拟事件。虚拟事件的形式是<<事件名称>>，可 利用构件的 event_add 方法来创建。例如，如果想为构件 w 创建一个新事件<<MyEvent>>， 该事件由鼠标右键或键盘上的 Pause 键触发，则执行下列语句：

```
w.event_add("<<MyEvent>>","<Button-3>","<KeyPress-Pause>")
```

此后就可以像系统定义的事件一样使用了。例如：

```
w.bind("<<MyEvent>>",myHandler)
```

在构件 w 上点击右键或按下 Pause 键都会触发函数 myHandler。

8.4 模型—视图设计方法

8.4.1 将 GUI 应用程序封装成对象

GUI 编程的一个常用技术是将整个应用程序封装成一个类，在应用程序类中建立图形界 面并处理各种交互事件。具体来说，GUI 应用程序类应该首先创建一个主窗口，并在其中布 置所需的各种构件，然后再为各个构件编写事件处理程序（都是类的方法）。这种做法的好处是：由于事件处理函数都定义为应用类的方法，而类的方法很自然地能访问类中的实例变量， 所以只要我们将界面中的各种构件也存储为实例变量，就能实现程序的处理代码与程序的图 形界面进行“无缝集成”。

在用 Tkinter 编程时，根据需要可以有多种方式来建立程序主窗口：

（1）在应用程序类中创建自己的根窗口，即程序自成体系。代码大致形如：

```
class MyApp:
    def __init__ (self):
        root = Tk()
        b = Button(root,...)
        ...
        root.mainloop()
app = MyApp()
```

（2）程序主窗口是程序类外部的某个窗口的子构件，该外部窗口在创建程序实例时作为 参数传递给构造器。例如：

```
class MyApp:
    def __init__ (self,master):
        f = Frame(master,...)
        b = Button(f,...)
        ...
        root = Tk()
app = MyApp(root) root.mainloop()
```

（3）将应用程序类定义为框架构件类的子类，即程序就是窗口，窗口就是程序。如：

```
class MyApp(Frame):
    def __init__ (self):
        Frame. __init__ (self) # 先用父类的构造器进行初始化
        b = Button(self,...)
        ...
app = MyApp() app.mainloop()
```

在应用程序类的设计中，如果一个构件具有“全局性”，即多个方法都要访问该元素，那 么就需用一个实例变量来存储（引用）这个构件，因为类的实例变量在所有类方法中都可访 问，而局部变量只在某一个方法中可见。

作为例子，我们定义一个应用程序类 MyApp，该程序的用户界面包括窗口、标签和按钮。 我们采用上述第一种方式，即程序创建自己的根窗口。根窗口和标签构件被存储为实例变量 self.root 和 self.t，以便 MyApp 类的所有方法都能引用它们；两个按钮则被存储为局部变量

b1 和 b2，这样在其他方法中是不能引用它们的。

【程序 8.11】myapp.py

```
from Tkinter import *
class MyApp:
    def __init__ (self):
        self.root = Tk()
        self.root.title("My App")
        self.t = Label(self.root,text="Spam")
        self.t.pack()
        b1 = Button(self.root,text="Play",command=self.changeText)
        b2 = Button(self.root,text="Quit",command=self.root.quit)
        b1.pack()
        b2.pack()
        self.root.mainloop()
        self.root.destroy()
    def changeText(self):
        if self.t["text"] == "Spam":
            self.t["text"] = "Egg"
        else:
        self.t["text"] = "Spam"
app = MyApp()
```

程序 8.11 定义了类 MyApp，在其构造器 init__中首先创建根窗口 root，然后添加一个 标签和两个按钮。点击按钮 b1 时的回调函数是类 MyApp 中自定义的方法 changeText（功能 是改变标签的文本），点击 b2 时的回调函数是根窗口的内建方法 quit（退出事件循环）。标签 构件必须作为实例变量存储，因为 init 和 changeText 方法都要引用它；而根窗口和两个按 钮在本例中既可以作为实例变量存储，也可以作为局部变量存储。创建各构件并完成布局之 后进入事件循环，等待处理事件。

类只是一个定义，封装成类的应用程序如何执行呢？我们通常会为应用程序类定义一个 专门的启动方法 run，将来创建应用程序对象后通过调用对象的 run 方法来启动程序功能。本 例中 MyApp 程序对象 app 一经创建就自动进入程序主循环，这是因为我们将所有启动代码 包括 mainloop 都放在构造器 init之中的缘故。程序启动后，点击 **Play** 按钮可以看到标签 内容在**“Spam”**和**“Egg”**之间切换。点击 **Quit** 按钮将退出事件循环，从而执行 **init**的最 后一条语句 root.destroy 关闭根窗口①。

作为练习，读者可以用上述第二、三种方式来改写程序 8.11。

8.4.2 模型与视图

复杂应用程序经常可以分解成两个部分：核心逻辑和用户界面。程序的核心逻辑部分称

为模型（model），它负责为应用问题建模，管理应用问题的数据和行为，并对来自用户界面的数据请求或数据更新指令进行响应。程序的用户界面部分称为视图（view），它负责显示模 型的当前数据状态，响应用户的交互动作。模型和视图是相互独立的，可以分开设计和测试， 从而简化程序结构、降低设计难度，这称为模型一视图（MV）设计方法。

模型与视图之间的桥梁称为控制器（controller）：用户通过用户界面发出交互动作，从而 触发事件处理器（回调函数）做出相应处理，导致模型的状态发生改变；模型状态的改变又 导致视图的更新，从而向用户输出结果。如果把模型和视图之间的控制器考虑进去，这种方 法也称为模型一视图一控制器（MVC）方法（图 8.25）。



图 8.25 模型一视图体系结构

用 MV 方法设计程序时，定义模型不需要用到视图中的元素，定义视图也不需要用到模 型中的数据，从而可以分别设计和测试。另外，同一模型可以使用不同的视图来达到不同的 目的，例如可以先设计一个基于文本界面的简单视图来测试模型的正确性，确定模型没有问 题后再去设计更美观易用的 GUI 视图。在实际开发中，经常将用户界面封装成界面对象，这样随时可以通过替换不同的界面对象来改变模型的外观和用户体验。 模型设计与视图设计在很多方面都不一样。为应用问题建立模型是“智力密集型”的工作，需要创造性的算法设计；而构造视图则是“劳动密集型”的工作，需要用户友好和美观。 模型的建立很难借助自动化的设计工具，而图形界面的构建大部分都可利用设计工具自动或 半自动地完成。

> ① 有的环境可能在退出事件循环时就自动关闭根窗口。

8.4.3 编程案例：汇率换算器

本节通过一个应用实例来介绍 MV 方法的具体应用。我们希望设计一个汇率换算器程序， 其功能是将外币换算成人民币，或者相反。最终的版本是图形用户界面的，但在设计过程中，我们还会设计一个文本界面的版本用来测试程序功能的正确性。

我们首先设计程序模型，这是由 CCApp 类实现的。设计 CCApp 时并不限定将使用的界 面，但随着 CCApp 的细化设计，我们会得到有关界面的功能需求，从而有助于程序用户界 面的设计和实现。

程序规格

汇率换算器程序用于在外币与人民币之间进行换算。 输入：外币币种，换算方向，金额 输出：等值的目标币种金额

明确候选对象

根据程序需求来确定对解题有用的对象。汇率换算器处理的是货币，货币用一个简单的 符号或数值就能表示，没有必要封装成对象。我们只将应用程序模型部分封装成一个类 CCApp，该类的实例需要记录当前的汇率信息，并至少需要实现构造器方法 init__和程序启 动方法 run。也许还需要若干辅助方法，这只有到设计主算法时才会明确。

除了核心部分，程序的另一个组成部分是用户界面。我们将界面也封装成对象，这样做 的好处是：在不改变模型的情况下，通过换用不同界面对象即可改变程序的外观。假设程序 将使用的界面对象是 xInterface，目前还不清楚这个类应有的行为，但随着我们精化 CCApp 的设计，就会明确需要从用户输入什么信息和向用户显示什么信息，所有输入和输出正对应 着 xInterface 类要实现的方法。

实现模型

由于本章重点是用户界面，所以作为例子的汇率换算器程序的模型很简单：按照存储的 当前汇率信息，将给定数额的外币换算成人民币，或将人民币换算成外币。对如此简单的问 题，只需一个 CCApp 类即可实现。当然，对于复杂程序，模型会涉及多种对象，那样就需 要实现多个类。模型的设计可采用自顶向下逐步求精方法，在此逐步细化的过程中，即可逐 步明确用户界面应该提供哪些方法。下面是 CCApp 类的定义：

【程序 8.12 之一】ccapp.py

```
class CCApp:
    def init (self, inter):
        self.xRate = {'USD':6.306,
            'Euro':8.2735,
            'Yen':0.0775,
            'Pound':10.0486}
        self.interface = inter
    def run(self):
        while True:
            to,fc,amount,bye = self.interface.getInfo()
            if bye:
                break
            elif to == 'RMB':
                result = amount * self.xRate[fc]
            else:
                result = amount / self.xRate[fc]
            self.interface.showInfo(result)
```

首先看构造器 __init__()，它负责汇率换算器的初始化，具体就是用一个字典 self.xRate 来存储若干外币名称及其对人民币的汇率①。注意， __init__方法还有个参数 inter，它代表程 序的用户界面，后面我们会分别用两种用户界面对象传递给此参数，从而得到两种版本的换 算器程序。

将来创建汇率换算器实例之后，通过调用实例的 run 方法来启动换算功能。换算器的核 心算法是个循环，每次循环完成一次换算，换算所需的各种信息都来自用户界面。可以看出 总体上换算过程仍然是简单的 IPO（即输入——处理——输出）算法模式，涉及的输入信息 包括换算方向、换算的外币、换算金额和退出标志（通过界面提供的 getInfo 方法获得），输 出信息就是换算结果（通过界面提供的 showInfo 方法显示）。当用户在用户界面选择退出时， 则不再循环，程序结束。

至此，我们实现了汇率换算器的核心功能，实现了程序的模型部分。但现在还无法测试 程序，因为还没有建立用户界面。但模型对用户界面的基本要求已经确定了，就是要提供 getInfo 和 showInfo 方法，参见图 8.26。

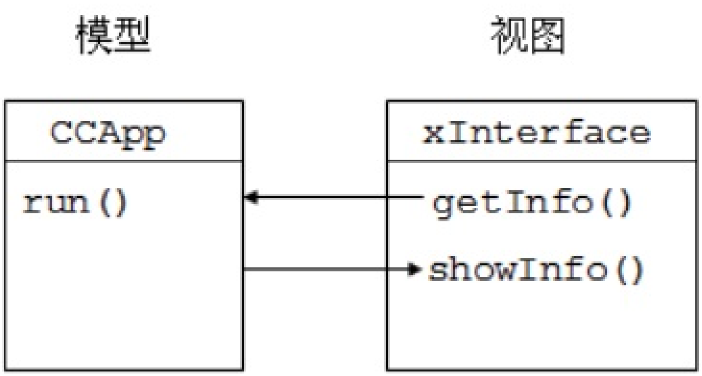


图 8.26 模型-视图方法例 基于文本的用户界面

CCApp 类的定义中用到了很多用户界面的功能，可见模型的设计实现过程也揭示了用户 界面应当如何设计。和模型一样，我们将视图（用户界面）的功能也封装成一个类，这个界 面类必须包括 CCApp 类中所用到的所有方法：quit、close、getCurr、getDirection、getAmount

和 display。如果以不同方式来实现界面类 CCInterface，就能产生具有不同外观的换算器程序， 注意作为基础的模型 CCApp 类是不变的。可见视图与模型可以独立地设计，为同一模型可 以设计多种视图。

> ① 程序中的汇率数据是 2012 年 4 月 18 日的汇率。

由于一般来说 GUI 比较复杂，为了尽快测试模型的正确性，可以先设计一个简单的文本界面。这个界面纯粹用于测试，无需过多考虑用户友好性，因此我们以最简单最直接的方式 来实现 CCApp 所需的各种界面功能。

【程序 8.12 之二】ti.py

```
class TextInterface:
    def __init__ (self):
        print "Welcome to Currency Converter!"
        self.qFlag = False # Quit flag
        self.fc = 'USD' # foreign currency selected
        self.to = 'RMB' # convert to?
        self.amt = 0 # amount to be converted
    def getInfo(self):
        self.qFlag = self.getQFlag()
        if self.qFlag:
            self.close()
        else:
            self.fc = self.getFC()
            self.to = self.getTo()
            self.amt = self.getAmount()
        return self.qFlag,self.fc,self.to,self.amt
    def getQFlag(self):
        ans = raw_input("Want to quit? (y/n) ")
        if ans[0] in 'yY':
            return True
        else:
            return False
    def getFC(self):
        return raw_input("Choose among {USD,Euro,Yen,Pound}: ")
    def getTo(self):
        ans = raw_input("Convert to RMB? (y/n) ")
        if ans[0] in 'yY':
            return 'RMB'
        else:
            return self.fc
    def getAmount(self):
        if self.to == 'RMB':
            return input("How much " + self.fc + "? ")
        else:
            return input("How much RMB? ")
    def showInfo(self,r):
        if self.to == 'RMB':
            print "%.2f %s ==&gt; %.2f RMB" % (self.amt,self.fc,r)
        else:
            print "%.2f RMB ==&gt; %.2f %s" % (self.amt,r,self.fc)
    def close(self):
        print "Goodbye!"
```

下面我们利用此用户界面来测试 CCApp 的正确性。为此目的，只需创建一个文本界面 对象，再创建 CCApp 对象，然后启动换算。测试程序如下：

【程序 8.12 之三】testti.py

```
from ccapp import CCApp
from ti import TextInterface
inter = TextInterface()
cc = CCApp(inter)
cc.run()
```

以下是测试运行示例，黑体部分是用户输入。结果表明程序的模型部分实现了预期的功 能。

```
Welcome to Currency Converter! Want to quit? (y/n) n
Choose among {USD,Euro,Yen,Pound}: USD
Convert to RMB? (y/n) y
How much USD? 100
## 100.00 USD ==&gt; 630.60 RMB
Want to quit? (y/n) n
Choose among {USD,Euro,Yen,Pound}: Euro
Convert to RMB? (y/n) n
How much RMB? 10000
## 10000.00 RMB ==&gt; 1208.68 Euro
Want to quit? (y/n) y
Goodbye!
```

实现 GUI

经过文本界面的测试，如果确信核心部分没有问题，即可转向设计更复杂但更加用户友 好的图形界面。我们要做的是确定图形界面的各种构件及布局，然后编写构件的处理代码。 与文本界面类似，图形界面需要提供的功能在模型设计过程已经确定了，图形界面必须支持 与文本界面相同的方法，另外也许还需要一些辅助方法。

根据模型部分所要求的界面功能来设计图形界面：选择要换算的外币种类，由于每次只 处理一种外币，故可用单选钮实现；输入和显示外币及等价人民币的金额，可用两个录入框 实现；双向换算和退出用三个命令按钮实现。至此即大致确定了图形界面的外观，接下来即 可为构件（主要是命令按钮）实现处理代码。最终得到如下 GUInterface 类定义：

【程序 8.12 之四】gui.py

```
from Tkinter import * class GUInterface:
def init (self): self.root = Tk()
self.root.title("Currency Converter")
self.qFlag = False # Quit flag
self.fc = StringVar() # foreign currency selected self.fc.set('USD')
self.to = 'RMB' # convert to?
self.amt = 0 # amount to be converted self.aRMB = StringVar() # amount of RMB self.aRMB.s
self.aFC = StringVar() # amount of foreign currency self.aFC.set('0.00')
Label(self.root,textvariable=self.fc).grid( row=0,column=0,sticky=W)
Label(self.root,text='RMB').grid( row=0,column=2,sticky=W)
self.e1 = Entry(self.root,textvariable=self.aFC) self.e1.grid(row=1,column=0,rowspan=2)
self.e2 = Entry(self.root,textvariable=self.aRMB) self.e2.grid(row=1,column=2,rowspan=2)
self.b1 = Button(self.root,
text='----&gt;',command=self.toRMB) self.b1.grid(row=1,column=1)
self.b2 = Button(self.root,
text='&lt;----',command=self.toFC) self.b2.grid(row=2,column=1)
self.f = Frame(self.root) self.f.grid(row=3,column=0,columnspan=3) self.r1 = Radiobutton(
variable=self.fc,value='USD') self.r1.grid(row=0,column=0)
self.r2 = Radiobutton(self.f,text='Euro',
variable=self.fc,value='Euro') self.r2.grid(row=0,column=1)
self.r3 = Radiobutton(self.f,text='Yen',
variable=self.fc,value='Yen') self.r3.grid(row=0,column=2)
self.r4 = Radiobutton(self.f,text='Pound',
variable=self.fc,value='Pound') self.r4.grid(row=0,column=3)
self.rate = Button(self.root,text='Update Rates') self.rate.grid(row=4,column=1)
self.qb = Button(self.root,text='Quit',command=self.close) self.qb.grid(row=5,column=1)
def getInfo(self): self.root.mainloop()
return self.qFlag,self.fc.get(),self.to,self.amt
def showInfo(self,r): rStr = "%.2f" % r if self.to == 'RMB':
self.aRMB.set(rStr) else:
self.aFC.set(rStr)
def toRMB(self): self.to = 'RMB'
self.amt = eval(self.aFC.get()) self.root.quit()
def toFC(self):
self.to = self.fc.get()
self.amt = eval(self.aRMB.get()) self.root.quit()
def close(self): self.qFlag = True self.root.quit() self.root.destroy()
```

这个类中的 getInfo 和 showInfo 是被模型部分调用的方法，用于输入和输出；其他几个 方法都是辅助方法，用来设置输入输出的信息。在此需要解释一下用到的技术性手段：当核 心程序调用界面的 getInfo 方法时，self.root.mainloop 方法使图形界面进入事件循环，从而能 够处理用户在界面上的各种交互事件（如在录入框中输入数据、点击单选钮选择货币、点击 换算按钮等）。当用户点击换算按钮，相应的处理程序 toRMB 和 toFC 在设置有关信息后必 须用 self.root.quit 方法来退出事件循环，从而使 getInfo 方法结束并将控制返回核心部分。

下面我们利用此图形用户界面来实现图形版的汇率换算器。和前面测试文本界面一样， 在主程序中先创建一个图形界面对象，再创建 CCApp 对象，然后启动换算器。程序如下：

【程序 8.12 之五】testgui.py

```
from ccapp import CCApp
from gui import GUInterface
inter = GUInterface()
cc = CCApp(inter)
cc.run()
```

执行此程序，在图形界面中选择 Euro，并在 RMB 录入框中输入 10000，最后点击“<----” 按钮，得到的结果如图 8.27 所示：

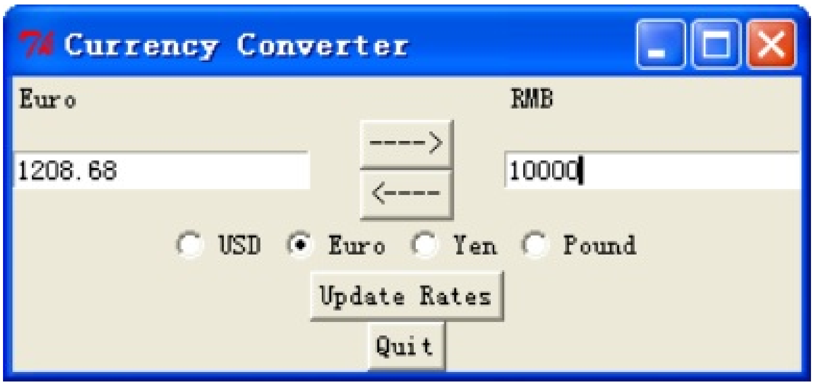


图 8.27 图形版汇率换算器

从图 8.27 还可看到，我们的图形界面中还有一个前面未提到的按钮 Update Rates，这是 用来更新汇率数据的，但本程序中没有为此按钮编写处理程序，作为练习，读者可以自己试 着完善这个功能①。另外，支持换算的外币种类也很容易扩充。

至此我们完成了一个汇率换算器程序。通过这个程序的设计，我们看到，即使是如此简 单的程序，它的 GUI 设计也相当复杂。一般而言，图形界面由很多构件组成，创建构件并进 行布局是枯燥而繁琐的工作；而为构件编写相应的处理程序通常都比较简单直接。

8.5 练习

1. 编程实现一个计算器。可参考 Windows 的计算器。

2. 编程实现一个简单的文本编辑器。可参考 Windows 的记事本。

3. 编程实现一个 GUI 版的学生信息管理系统（参见 6.6 练习第 9 题）。

> ① 例如可以新开一个顶层窗口，在其中输入新的汇率数据，并更新 CCApp 实例的汇率字典 self.xRate。

第 9 章 模拟与并发

迄今为止，本书所讨论的计算具有两个特点：第一，计算是确定的，即只要输入相同， 程序执行后得到的结果总是一样的；第二，程序在任意时刻只做一件事，不能同时做多件事。 这是传统程序的典型特征。本章将介绍两种不属于这种典型形式的计算形式：一种是能够处 理随机现象的模拟方法，一种是能够同时做多件事的多线程并发。这两种计算形式的共同特 点是不确定性，即针对同样的输入，同一程序可能有不同的执行过程和结果。

9.1 模拟

现实中有很多问题，如果不利用计算机的话，就很难解决甚至不可能解决。例如天气预报，古人只能通过肉眼看天来做预测，现代人则通过为大气过程建立数学模型并进行数值计算来做预测，最新的理论更将确定性模型发展到不确定性模型，从而能对大气这个混沌系统的行为做出更准确预报。这一切都有赖于计算机模拟（simulation）技术的应用，即利用计算机为现实问题甚至假想问题建立模型，通过改变一些变量值或条件，来研究系统的行为，获得原本无法获得的信息。模拟是利用计算机解决现实问题时的一个强大技术，除了天气预报，还广泛用在风险分析、飞机设计、电影特效等很难直接解决真实问题的领域。

9.1.1 计算机建模

利用计算机解决现实中的问题，首先需要在计算机中将问题表示出来，这个过程称为建模（modeling），即建立描述现实问题的一个模型（model）。打个比方，用照相机拍摄自然景物就是建模，即得到自然景物在照相机中的表示（数字图像）。不过照相机“建模”追求 的是模型必须反映自然景物的每一个细节，最好是一模一样。而用计算机为现实问题建模， 追求的是模型必须抽象出问题的关键特征，至于非关键的部分则可以忽略。如此得到的模型 比较简单，虽然不一定和现实很“像”，但足够支持解决问题。现实问题在计算机内的模型 通常都是数学模型，即利用数学公式或数学过程来描述现实问题。

下面我们写一个简单的程序，该程序的行为具有“混沌”现象的特征。所谓混沌现象， 是指在确定性系统中发生的看上去随机、不规则的运动，即用确定性理论描述的系统却表现 出不确定的行为。混沌现象的特征是不可预测性和对初始条件的极端敏感性。读者想必听说 过著名的“蝴蝶效应”：某处的一只蝴蝶扇动一下翅膀，这一扰动有可能导致很远的另一个 地方的天气出现非常大的变化。这个比喻想说的其实就是气象具有混沌行为。事实上，现实 生活和工程技术问题中，混沌现象是无处不在的。下面的程序虽然简单，却具有混沌现象不 可预测和对初始值敏感的两个特征。

【程序 9.1】chaos.py

```
def main():
    x = input("Enter a number between 0 and 1: ")
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print x
main()
```

运行这个程序，可得如下输出：

```
Enter a number between 0 and 1: 0.2
0.624
0.9150336
0.303213732397
0.823973143043
0.565661470088
0.958185428249
0.156257842027
0.514181182445
0.974215686851
0.0979659811419
```

再次运行这个程序，但换一个输入数据 0.21，可得如下输出：

```
Enter a number between 0 and 1: 0.21
0.64701
0.89071343361
0.379637749907
0.918500422135
0.291943847024
0.806179285114
0.609391556931
0.928330600362
0.259478297495
0.749382311434
```

从运行结果可以发现，尽管程序的代码是确定的，但输出的 10 个结果毫无规律，好像 完全是不可预测的。此外，比较两次运行的输出结果，可以发现初始输入数据的微小变化会 使输出结果很快变得显著不同。这两点正是混沌现象的特征。本程序之所以能够显现出混沌 特征，是因为程序中使用了计算公式 k*x*(1-x)，反复利用这个公式求值就会导致混沌。 换句话说，程序 chaos 利用数学公式 k*x*(1-x)为混沌现象建立了模型。

一旦用计算机程序为现实问题建立了模型，我们就可以通过运行程序并分析结果来探索 现实问题的性质。这时，模型的好坏是至关重要的，错误的程序自然会给出错误的结果，但 正确的程序也可能因为不准确的模型而产生错误的结果。

9.1.2 随机问题的建模与模拟

现实中有许多不确定的事件，称为随机事件。例如抛一枚硬币，结果是正面朝上还是反

面朝上，这是不确定的。研究随机事件的数学方法是统计，例如经过大量统计试验可以得出结论：抛硬币时正面朝上和反面朝上的可能性是相等的，各占 50%。注意，说硬币正面朝 上和反面朝上的可能性各占 50%，并不意味着抛硬币试验将得到“正，反，正，反，…” 的结果，完全有可能出现一连串的正面或一连串的反面。

既然现实问题中存在随机事件，用计算机解决这类问题时就需要为随机事件建模，即程 序能够模拟随机事件的发生。例如，假设程序 P 能够模拟抛硬币并显示每次抛硬币的结果（“正”或“反”），则 P 应该具有这样的特性：每一次显示的结果是不可预测的，但多次运 行 P 之后“正”、“反”出现的次数应该是差不多的。可以设想 P 中有这样的语句：

```
if 模拟抛硬币的结果是正面:
    print "正"
else:
    print "反"
```

这里 if 后面的条件必须有时为真有时为假，但无法预测每一次运行时的真假；而多次运行后，条件为真为假的次数应该基本相等。

我们知道，计算机是确定性的机器，它总是按照预定的指令行事。对于一个程序，只要 程序的初始输入是一样的，那么程序的运行结果就是确定的、可预测的。就拿程序 9.1 来说，尽管它产生了看上去不可预测的数值序列，但那并非真正的随机，因为按照程序 9.1 中的数 学式去计算，从相同的 x 开始必能得到完全一样的数值序列。所以，程序 9.1 所用的数学式 不能用于模拟随机事件，我们需要更好的能产生随机数的方法。

随机数生成函数

计算机科学家对随机数生成问题有很成熟的研究，他们设计了一些数学公式，通过计算

这些公式就能得到随机数。不过，既然是用确定的数学公式计算出来的数值，那就不可能是数学意义上的随机，因此准确的名称实际上是“伪随机数”。伪随机数在实际应用中完全可 以当作真随机数来使用，因为它具有真随机数的统计分布特性，简单地说就是“看上去”是 随机的。

Python 语言提供了一个标准库模块 random，该模块中定义了若干种伪随机数生成函数。 这些伪随机数生成函数的计算与模块导入的时间（精度非常高）有关，因此每次运行函数所 产生的数值是不可预测的。random 模块中用得最多的随机数生成函数是 randrange 和 random。

randrange 函数生成一个给定范围内的整型伪随机数，范围由 randrange 的参数指定，具 体格式和 range 函数一样。例如，randrange(1,3)随机地产生 1 或 2，randrange(1,7)返回范围[1,2,3,4,5,6]中的某个数值，而 randrange(2,100,2)返回小于 100 的某个偶数。例如：

```
>>> from random import randrange
>>> for i in range(20):
print randrange(1,3),
1 2 2 2 1 1 2 2 2 1 1 2 1 2 2 1 1 1 1 1
>>> for i in range(20):
print randrange(1,7),
1 5 1 5 2 3 2 2 3 2 4 2 6 3 1 2 1 1 1 5
```

注意，由于试验次数较少（20 次），randrange 所生成的数值并未如我们期望的那样均匀分布， 但随着试验次数的增加，会发现 randrange 产生的值都具有差不多的出现次数。

random 函数可用来生成浮点型伪随机数，确切地说，该函数生成[0,1)区间中的浮点数。

random 不需要提供参数。例如：

```
>>> from random import random
>>> for i in range(5):
print random()
0.35382204835
0.997559174002
0.672268961152
0.889307826404
0.246100521527
```

注意，random 模块名与 random 函数名恰巧相同，不要因此而误用。

模拟

有了随机数生成函数，我们就可以来模拟随机事件了。以抛硬币问题为例，前面我们给 出了如下形式的代码：

```
if 模拟抛硬币的结果是正面:
    print "正"
else:
    print "反"
```

并且指出 if 后面的条件的真假应该是不可预测、均匀分布的。

考虑 randrange(1,3)：该函数产生随机数 1 或 2，每一次调用到底生成什么值是不可预测的，并且大量调用后两个数值出现的机会是一样的。据此，randrange(1,3) == 1 正是我们所需要的条件，此条件每一次计算时的真假是随机的，但长远来看真假情形各占 50%。将这 个条件代入上面的条件语句，即得

```
if randrange(1,3) == 1:
    print "正"
else:
    print "反"
```

这样，我们就通过调用合适的随机数生成函数的方式模拟了随机事件，这种模拟方法称为蒙特卡洛方法。 类似地，掷骰子也是现实中常见的随机问题，如果希望在程序中模拟掷骰子，可以这样做：

```
value = randrange(1,7) print "你掷出的点数是:",value
```

再看个例子，两个运动员打乒乓球，谁能赢呢？胜负自然取决于球员的技术水平，但又 并非水平高的人必然赢，毕竟体育比赛和天时地利人和等各种因素有关。既然比赛结果有随 机性，我们就可以利用蒙特卡洛方法来模拟比赛。假设 A、B 两个球员相互之间的胜率大致 是 55%对 45%，那么他们打一次比赛（比赛的单位可以是 1 分、1 局或 1 盘，在此并不重要）的结果可以用如下代码模拟：

```
if random() &lt; 0.55:
    print "A wins."
else:
    print "B wins."
```

这里，由于 random()生成的随机数均匀地分布在[0,1)区间内，所以有 55%的值落在 0.55 的左边，即 random() < 0.55 为真的可能性为 55%，为假（即 random() >= 0.55）的可能性为 45%， 这就恰当地模拟了 A 和 B 的胜率。注意，random() = 0.55 时应该算作 B 赢，因为 random() 生成的随机数包含 0 但不包含 1。

9.1.3 编程案例：乒乓球比赛模拟

众所周知，中国乒乓球项目的技术水平世界第一，以至于所有比赛的冠军几乎都由中国球员包办。为了增强乒乓球运动的吸引力，提高其他国家的人对这项运动的兴趣，国际乒联 想了很多办法来削弱中国球员的绝对优势，例如扩大乒乓球的直径、禁用某些种类的球拍、 改变赛制等等。在本节中，我们将编写程序来模拟乒乓球比赛，以便研究一项针对中国球员 的规则改革是否真的有效。这项改革是：从 2001 年 9 月 1 日起，乒乓球比赛的每一局比分从 21 分改为 11 分。

球员技术水平的表示 乒乓球是两个球员之间的比赛，比赛开始后由一个球员发球， 另一个球员将球接回来，然后两人交替击球，直至一方没能将球回到对方台上，这时另一方就得一分。一个球员有几 次发球机会，用完这些发球机会后将换发球。

比赛胜负由球员的技术水平决定，我们用两个球员对阵时各自的得分概率来表示他们的 技术水平。如果球员 A 与 B 水平相当，则 A 拿下 1 分的概率是 50%，B 拿下 1 分的概率也 是 50%；如果 A 水平较高，拿下 1 分的概率是 55%，则 B 拿下 1 分的概率就只有 45%了。 顺便指出，球员技术水平的表示方法并无一定之规，是由编程者自己主观确定的，关键 是表示方法要比较符合实际。我们这里采用的得分概率表示方法很简单，但没有考虑球员作 为发球方和接发球方的区别。读者可以考虑其他的表示方法，如：将球员的世界排名换算成 获胜概率，或者用球员作为发球方时的得分概率，或者用综合考虑发球得分概率和接发球得分概率的某个概率计算公式，等等。

模拟一回合比赛与得分

设 A、B 两球员比赛时，各自得分的概率为 prob 和 1-prob。利用蒙特卡洛方法，下面 的代码即模拟了得到 1 分的一回合比赛，这是整个模拟程序的核心功能。

```
if random() < prob:
    pointA = pointA + 1
else:
    pointB = pointB + 1
```

我们可以立刻来测试这个核心功能。假设 A 的得分概率是 0.55，让 A、B 进行 10000分的较量，看看各自得分情况如何。测试代码如下：

```
>>> from random import random
>>> pointA = pointB = 0
>>> for i in range(10000): if random() &lt; 0.55:
pointA = pointA + 1 else:
pointB = pointB + 1
>>> print pointA,pointB 5430 4570
```

最后得分差不多就是 55%比 45%，可见模拟比赛的结果确实反映了 A、B 双方的实力。

模拟一局比赛

乒乓球比赛不是按比赛回合来判定胜负的，而是采用将若干回合组成一局的方式，以局 为单位来判定胜负。老规则采用每局 21 分制，新规则采用每局 11 分制。我们利用上述模拟一回合比赛及得分的代码，改成以局为单位进行比赛（假设采用 21 分制）。

```
>>> def oneGame():
pointA = pointB = 0
while pointA != 21 and pointB != 21: if random() &lt; 0.55:
pointA = pointA + 1 else:
pointB = pointB + 1 return pointA, pointB
```

函数 oneGame 模拟了 21 分制的一局比赛：只要还没人达到 21 分，就继续进行回合较量；

否则退出循环，并返回本句中 A 和 B 各自的得分 pointA 和 pointB。调用 oneGame 的人可以比较这两个返回值的大小，以判断是谁赢了这一局。

下面我们来测试 oneGame 函数，让两个球员进行 1000 局较量，看看胜负如何。

```
>>> gameA = gameB = 0
>>> for i in range(1000):
        pointA, pointB = oneGame()
        if pointA > pointB:
            gameA = gameA + 1
        else:
            gameB = gameB + 1
>>> print gameA,gameB
751 249
```

出人意料的是，虽然 A、B 在每一回合的获胜概率相差不大（55%比 45%），但如果按每局 21 分制进行比赛的话，A 的胜局数遥遥领先于 B（75%比 25%）！这里面的道理是显然的，每回合的胜负偶然性对 21 分一局的胜负来说影响减小了，得分能力稍强的人更加可能赢得一局。可以想象，如果将一局的得分减少，就像国际乒联所做得那样改成每局 11 分，那么 每回合的胜负偶然性对一局的胜负就会有较大影响。稍后我们将在程序中验证这一点。

模拟一场比赛

一场乒乓球比赛也不是无限制地打很多局才能定胜负，一般都是采取 3 局 2 胜、5 局 3 胜或 7 局 4 胜的方式来完成比赛。下面我们采用 21 分制、3 局 2 胜的赛制，来编写模拟一 场比赛的程序 oneMatch，并通过模拟 100 场比赛来测试 oneMatch。

```
>>> def oneMatch():
        gameOver = [(3,0),(0,3),(3,1),(1,3),(3,2),(2,3)]
        gameA = gameB = 0
        while not (gameA,gameB) in gameOver:
            pointA, pointB = oneGame()
            if pointA > pointB:
                gameA = gameA + 1
            else:
                gameB = gameB + 1
        return gameA, gameB
>>> matchA = matchB = 0
>>> for i in range(100):
        gameA, gameB = oneMatch()
        if gameA > gameB:
            matchA = matchA + 1
        else:
            matchB = matchB + 1
>>> print matchA,matchB
89 11
```

可见按 3 局 2 胜来计算胜负，导致 A 和 B 的胜负更加悬殊了（89%比 11%），这是因为 3 局 2 胜的规则将每一局胜负偶然性的影响削弱了。

完整程序

通过以上设计过程，我们的模拟乒乓球比赛的程序越来越完善了。接下去我们可以进一步改善程序的功能，例如将球员的技术水平改成由用户输入而不是固定的 0.55，允许采取不同的比赛规则（21 分或 11 分），增加对比赛结果的分析，等等。这些新增特性就不详细解释了，请读者自行阅读下面的完整程序代码。

【程序 9.2】pingpong.py

```
from random import random
def getInputs():
    p = input("Player A's winning prob: ")
    n = input("How many matches to simulate? ")
    return p, n
def simNMatches(n,prob,rule): matchA = matchB = 0
    for i in range(n):
    gameA, gameB = oneMatch(prob,rule)
    if gameA &gt; gameB:
        matchA = matchA + 1
    else:
        matchB = matchB + 1
    return matchA, matchB
def oneMatch(prob,rule):
    gameOver = [(3,0),(0,3),(3,1),(1,3),(3,2),(2,3)]
    gameA = gameB = 0
    while not (gameA,gameB) in gameOver:
        pointA, pointB = oneGame(prob,rule)
        if pointA > pointB:
            gameA = gameA + 1
        else:
            gameB = gameB + 1
        return gameA, gameB
def oneGame(prob,rule):
    pointA = pointB = 0
    while not gameOver(pointA,pointB,rule):
        if random() < prob:
            pointA = pointA + 1
        else:
            pointB = pointB + 1
    return pointA, pointB
def gameOver(a,b,rule):
    return (a&gt;=rule or b&gt;=rule) and abs(a-b)&gt;=2
def printSummary(a,b): n = float(a + b)
    print "Wins for A: %d (%0.1f%%)" % (a, a/n*100)
    print "Wins for B: %d (%0.1f%%)" % (b, b/n*100)
def main():
    p, n = getInputs()
    matchA, matchB = simNMatches(n,p,21)
    print "\nRule: 21 points, best of 3 games." printSummary(matchA,matchB)
    matchA, matchB = simNMatches(n,p,11)
    print "\nRule: 11 points, best of 3 games." printSummary(matchA,matchB)
main()
```

本程序的核心代码在前面介绍了，其他如 getInputs 和 printSummary 的功能都是显然的， 只有判断一局比赛结束的 gameOver 中有个条件 abs(a-b)>=2 需要说明一下。乒乓球比赛规 则规定，赢得一局比赛的球员至少要比对手多得 2 分，即 20:20（或 10:10）之后，一定要 连得 2 分才能赢得此局。

下面是本程序的一次运行结果：

```
Player A's winning prob: 0.52
How many matches to simulate? 100
Rule: 21 points, best of 3 games. Wins for A: 74 (74.0%)
Wins for B: 26 (26.0%)
Rule: 11 points, best of 3 games. Wins for A: 68 (68.0%)
Wins for B: 32 (32.0%)
```

结果表明，假设球员 A 对球员 B 的得分概率为 52%，当采用每局 21 分、3 局 2 胜的规 则时，A 有 74%的机会赢得比赛；当采用每局 11 分、3 局 2 胜的规则时，A 的获胜概率降到了 68%。这说明，国际乒联对规则的修改确实能削弱强手的优势程度。不过即便如此， 强手获

胜的概率还是相当高，何况中国球员的得分概率恐怕远不止 52%。或许要将每局分 数再减少点，或者用其他方法，才能增加外国选手获胜机会吧:-)。

读者可以试着修改程序 9.2，比如采取发球方得分概率作为技术水平的表示，并且将发 球、换发球等因素添加到模拟程序中。

9.2 原型法

我们在 4.3 中介绍了自顶向下逐步求精的程序设计方法。自顶向下设计是非常强大的程 序设计技术，但它也有不适用的场合。

自顶向下设计的第一步是顶层设计，这需要设计者对问题的全局有清晰的认识。万一要 解决的问题非常复杂，或者用户需求不是很完整、清晰，这时顶层设计就非常困难。另外， 设计者有时候会卡在自顶向下层次中的某一层，这就导致下层的精化无法继续，从而影响整 个程序的开发。即便前面这两个问题都不存在，自顶向下设计也存在开发周期过长、工作量 太大的缺点。

另一种程序设计方法是原型法（prototyping）。这种方法的思想是，先开发一个简单版 本，即功能少、界面简单的版本，然后再对这个简单版本逐步进行改善（添加或修改功能）， 直至完全满足用户需求。初始精简版程序称为原型（prototype）。应用原型法来进行软件开 发的步骤大致如下：

（1）确认基本需求；

（2）创建原型；

（3）向用户演示或交付用户试用，获得反馈意见；

（4）改善原型；回到（3），重复（3）、（4），直至用户最终认可。 可见，原型技术不是对整个问题按照“设计、实现、测试”的过程来开发，而是先按此

过程创建一个原型，然后根据用户反馈再重复此过程来改善原型。这样，通过许多“设计一 实现一测试”小循环，原型逐步得到改善和扩展，直至成为最终产品。因此原型法也称为“螺 旋式开发”。原型法适合于大型程序开发中需求分析难以一次完成的场合，也常用在程序用 户界面设计领域。

原型法开发有很多优点，例如：用户可以通过实际使用的体验来评价软件产品是否合意， 而不是仅凭开发者口头的描述；开发者和用户可以在开发过程中发现以前没有考虑到的需求 或问题；开发者可以在产品开发的早期就获得用户反馈，避免在开发后期来修改设计，因为 越往后，修改的代价就越大。

原型法开发中的原型可以有两种处理方法。一种方法是，一开始构造的原型就是最终产 品的核心，后续工作都是对原型的改善，通过不断的积累和修改最后得到符合用户需求的产 品。开发过程中，原型尽管功能不完善，但一直是可用的，甚至可以作为在最终产品交付前 的替代产品。另一种方法是，初始创建的原型只是作为与用户进行交互的工具，通过不断展 示给用户看而获得反馈并改进。等到用户认可原型，该原型即被丢弃，开发者将基于用户已 经确认的需求开始正式开发产品。这种做法称为“快速原型法”，因为其主要目的是尽快构 造系统模型，强调的是开发速度。

我们在 9.1.3 中就采用了原型法来设计实现乒乓球比赛的模拟程序。 解决问题的关键是模拟乒乓球比赛，而比赛的最基本动作是打一个回合及记分，因此我们一开始就考虑如何模拟一个回合的比赛并给胜方得分。即：

```
if random() < prob:
    pointA = pointA + 1
else:
    pointB = pointB + 1
```

经过测试，可以确认这段代码确实能够模拟具有特定得分概率的球员之间的比赛。在此 基础上，根据实际乒乓球比赛的规则要求，我们将回合扩展到局，又扩展到一场 3 局 2 胜的 比赛。在设计的前期阶段，我们做了很多简化，比如直接假设球员的得分概率为 0.55，直接 规定采用 21 分一局的规则等等。随着螺旋式开发的进展，程序功能越来越完善，所有被简 化的东西最终都会以完善的形式实现。总结这个开发过程，我们的模拟程序大致经历了如下 阶段：

阶段 1：建立原型，能进行回合比赛并为胜方记分；

阶段 2：扩展为能进行一局比赛；

阶段 3：扩展为能进行 3 局 2 胜的比赛；

阶段 4：球员的得分概率、比赛次数改为由用户输入；

阶段 5：添加结果分析，输出分析结果。

通过原型法来编程序，对初学程序设计的人来说是很合适的，因为可以从原型获得最终 程序的感性认识，并进而理解自己到底要写什么样的程序。

要指出的是，螺旋式开发并不是用来取代自顶向下设计的，这两种设计方法是互为补充 的关系。例如，对于大型的复杂程序，如果用原型法，可能原型本身也比较复杂，这时就可 以采用自顶向下设计来创建原型。好的设计者应该根据情况选用多种设计方法，这一切都要 通过实践来学习掌握。

9.3 并行计算*

计算思维是建立在计算机的能力和限制之上的，计算机科学家的任务是尽量发扬计算机 的能力，避开计算机的限制。传统的计算概念是在计算机发明之初形成的，就是由一个处理 器按顺序执行一个程序的所有指令。并行计算则突破了这种限制，试图让多个处理器同时做 事情。并行计算的好处是显然的，想想一个人吃一锅饭与一百个人同时吃一锅饭的差别，就 能理解并行计算的威力。

可以在不同层次上讨论并行。最底层是机器指令级并行，但这不是我们要关心的层次， 本节主要讨论在程序层次上的并行。

9.3.1 串行、并发与并行

计算机执行程序时，如果采用按顺序执行的方式，即仅当一个程序执行完毕，下一个程序才能开始执行，则称为串行（serial）执行。在串行执行方式下，CPU 每次由一个程序独 占使用，只要当前程序还没有结束，下一个程序就不能使用 CPU。这就像排队买东西，营 业员（即 CPU）每次只为一个顾客服务，等前面的顾客走了，后面的顾客才能获得服务。 串行执行方式有一个缺点，即 CPU 的利用率不高。例如，当一个程序正在等待用户输 入，这时 CPU 会在相当长的时间内无事可做；但因为程序还没结束，所以 CPU 又不能去执行别的程序。

为了提高 CPU 的利用率，计算机可以采用这样的执行方式：当程序 P1 因为等待输入或 其他原因而暂时不用 CPU 时，CPU 就去执行另一个程序 P2；当 P1 结束输入时，CPU 再回 去继续执行 P1。

其实这种执行方式并不稀奇，现实中就有很多类似的做法。例如，在邮局的服务窗口前 有多人排队，如果第一个顾客要寄特快专递，需要填写很多信息，这时营业员就处于空闲状 态。假设排在第二的顾客只是想买张邮票，这时如果营业员严格按照排队原则讲究先来后到 的话，那么即使没事做也不能为后面的顾客服务，非得等到第一个顾客的事情处理完毕才接

待第二个顾客。这种方式显然是非常低效的，服务质量很差。相反，如果营业员能够趁第一个顾客填写信息时抽空为后面的顾客办理业务，就能大大提高服务效率。又如，厨师做菜时， 如果一个菜正在锅里煮着，厨师肯定会去处理第二个菜，而不是非要等到第一个菜做好了才 去做第二个菜。

那么，计算机能不能实现上述执行方式呢？计算机程序的执行是由操作系统控制的，而现代操作系统都支持"多道程序"或"多任务"，即允许多个程序同时执行。相信读者都有同时运行多个程序的经验：一边打开浏览器 浏览网页，一边打开 MP3 播放器听音乐，一边还挂着 QQ 聊天程序。注意，这里所谓的"同 时"是在宏观意义上使用的。在微观上，一个 CPU 不可能真正"同时"执行程序，因为 CPU 在任一时刻只能执行一条指令。操作系统其实是在让多个程序分时使用 CPU，即 CPU 一会 儿执行 P1 程序的若干条指令，一会儿又转而执行 P2 程序的若干条指令，然后又回到 P1 继 续执行它的指令。总之，CPU 不停地在多个程序之间切换执行。由于 CPU 的运算速度非常 快，用户感觉不到这种切换过程，因此从宏观上看，就好像多个程序在同时执行一样。这种 多个相互独立的程序交叉执行的方式称为并发（concurrent）执行。并发执行的多个程序的 起止时间是交叉重叠的，而不是串行执行方式下的前后相继。

当然，如果计算机系统中有多个处理器（核心是 CPU），那么就可以做到真正的多个程序"同时"执行，因为各 CPU 可以在同一时刻执行各自的指令。为了与单一 CPU 上的并发 相区别，我们称这种执行方式为并行（parallel）执行。事实上，当今计算机技术的一个发 展方向就是多处理器并行计算。例如，中国的"天河一号"计算机曾经在全球最快计算机排 名中名

列第一，它拥有 6144 个通用处理器和 5120 个加速处理器，其二期产品“天河一号 A”更是拥有 14336 个六核处理器、7168 个加速处理器和 2048 个八核处理器。即使在个人计算机领域，如今也普遍采用多核处理器，例如市场上已经有了 8 核处理器。

9.3.2 进程与线程

操作系统控制处理器在多个程序之间切换执行的过程称为调度。传统的多任务操作系统是以进程为单位进行调度的。进程（process）是指程序的一次执行所形成的实体，每当程序 开始执行，就会创建一个进程。每个进程由程序代码以及一些状态信息（如进程数据的当前 值和当前执行点等）组成，状态信息也称为进程的上下文。

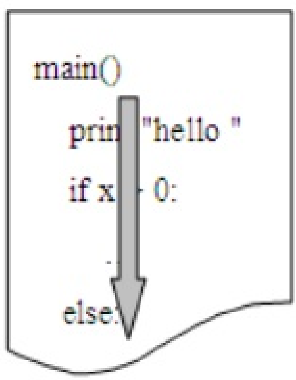
注意，程序与进程是不同的概念。首先，不同程序在计算机中执行，自然形成不同的进 程。其次，即使是同一个程序，当它在计算机中多次执行时，也会创建多个不同进程，这些 进程虽然具有相同的程序代码，但各有自己的上下文。例如，在 Windows 中我们可以同时 运行多个记事本程序（notepad.exe），以便编辑多个文本文件。这时，在系统中就创建了多 个 notepad 进程①。

通常，操作系统通过划分时间片来调度进程，各进程分时占有处理器来执行指令。从宏 观上看，多个进程是在同时运行。这就是操作系统的多任务机制。如前所述，多进程并发执 行可以提高系统资源的利用率。

但是，多任务、多进程也有缺点。第一，实现多进程并发需要花费不少系统开销。因为 每创建一个进程都要为它分配一些内存，以便存储它的上下文；而操作系统在不同进程间进 行调度时需要保存和恢复进程上下文。第二，进程间通信比较困难。进程和进程是隔离的， 各进程拥有自己的地址空间，一个进程不能访问另一个进程的地址空间，从而在进程之间很 难共享信息。因此，对于两个需要传递信息的进程，相互通信很麻烦。

> ① Windows 用户可以通过任务管理器来查看所有已创建的进程。

为了解决上述两个问题，产生了线程的概念。传统程序是从第一行指令一直执行到最后 一行指令，程序中只有一个控制流。这就像写小说时，沿着唯一的故事线索推进。而所谓线程（thread），是指程序中的一段代码，它构成程序中一个相对独立的执行单位。线程概念 使我们可以从一个程序中分出多个控制流来执行多个任务，所以线程实际上是程序内部的多 任务机制。就好比一部小说在叙述过程中，同时存在着多个故事线索，多头并进。图 9.1 给 出了单线程与多线程的示意图。



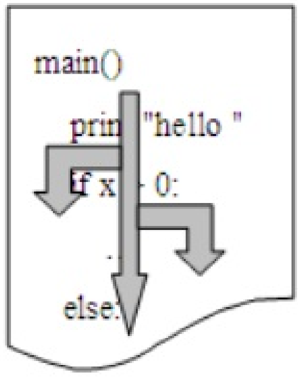


图 9.1 单线程程序与多线程程序

线程的字面意义就是程序（在执行时就是进程）的一个执行“线索”，一个程序中可以

有多个执行线索。《三国演义》里讲到庞统处理公务的故事，他“手中批判，口中发落，耳 内听词”，不到半天就把积攒了一百多天的事情都处理掉了。用本节术语来说，庞统就是采 用了多线程并发执行技术，所以能够高效率地解决问题。

线程与进程既相似，也有明显的区 别。系统中的多个进程一般是由执行不同程序而创建 的，而多个线程是同一程序（进程）的多个执行流；多个进程的状态是各自独立的，而多线 程可以共享地址空间（代码和上下文）；调度进程时上下文切换比多线程的上下文切换开销 大；进程间通信比较麻烦，而线程之间通过共享内存很容易通信。

在单个处理器上，可通过分时抢占或者线程自身执行情况来实现多线程的并发执行。前 者由操作系统进行调度，后者由线程自己放弃控制（比如执行到一个休眠指令时）。不同线 程之间切换得非常快，因此在用户看来各线程是在同时执行的。可见多线程与多任务、多进 程机制的实现技术是类似的。

在多处理器或多核处理器系统中，各处理器或内核都可以执行线程，从而多线程可以真 正地并行执行。由于多核处理器现已成为主流，因此了解并利用多线程技术对程序设计来说 变得越来越重要。

9.3.3 多线程编程的应用

线程原本是操作系统中的概念，是操作系统用于实现系统功能的工具。现在线程已演变成为用户程序可使用的工具，广泛用于应用程序设计。 多线程技术主要用于需要并发执行的场合。例如在很多游戏程序中，都需要维持一个动画场景，而玩家可以通过鼠标或键盘来输入操作指令，控制游戏的进行。假如程序只有一个 控制流，则当程序执行到等待用户输入指令的时候，由于用户输入较慢（相对 CPU 速度来 说），程序无法向前继续执行语句，因而无法更新动画画面，效果上动画就停顿了。如果将 等待用户输入的任务和维持动画的任务用两个线程来独立实现，则可解决这个问题。

上述例子其实具有一般性。如果在一个长时间计算任务（维持动画）期间需要对输入输 出事件（鼠标或键盘指令）做出反应，单线程程序是不行的，因为程序会阻塞在长时间计算 任务上，没有机会来检查输入输出。而如果用一个线程来执行这个长时间计算任务，并让另 一个线程监控输入输出事件，两个线程的并发执行就可以使应用程序在执行计算任务的同时 能对用户输入作出反应。如图 9.2 所示。

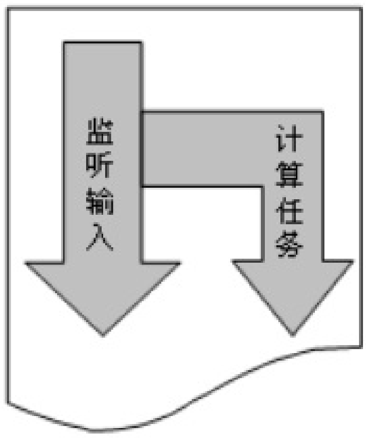


图 9.2 多线程的应用

此外，在科学计算中，很多数值算法都可以并行化，如矩阵的并行计算、线性方程组的 并行求解、偏微分方程的并行求解和快速傅里叶变换的并行算法等等。这时可以为每个处理 器建立一个线程，从而高效地进行计算。

虽然多线程技术有很多用途，但掌握多线程编程有点难度，即使对职业程序员也是如此。 例如，多线程技术涉及所谓竞态条件（race condition），即因为未曾预料到的、对事件之间 相对时序的严重依赖而导致的异常行为。具体例子如：两个客户同时登录订票网站，都看到 某航班还剩一个座位，于是都下单预定该座位，最终必然会因为谁先来后到而引起纠纷。如 果两人是在售票点排队购票（串行执行）就没有这个问题。因此，多线程程序与串行程序是 不同的，需要分析并协调各线程间的复杂的执行时序，这导致编程和调试都很困难。

多线程程序中，由于各线程是相互独立的，它们的并发执行没有确定次序可言，因此线 程是一种非确定性的计算模型，同一个程序的多次执行未必导致同样的结果。所以，多线程 计算模型不具有串行计算模型的可理解性、可预测性和确定性性质，使用时要非常小心。

9.3.4 Python 多线程编程

很多编程语言都支持多线程编程，Python 语言亦然。与其他编程语言相比，Python 的 多线程编程是非常简单的。

Python 提供了两个支持线程的模块，一个是较老的 thread 模块，另一个是较新的 threading 模块。其中 threading 采用了面向对象实现，功能更强，建议读者使用。

thread 模块的用法

任何程序一旦开始执行，就构成了一个主线程。在主线程中随时可以创建新的子线程去

执行特定任务，并且主线程和子线程是并发执行的。 每个线程的生命期包括创建、启动、执行和结束四个阶段。 当主线程结束退出时，它所创建的子线程是否继续生存（如果还没有结束的话）依赖于系统平台，有的平台直接结束各子线程，有的平台则允许子线程继续执行。

thread 模块提供了一个函数 start_new_thread 用于创建和启动新线程，其用法如下：start_new_thread(<函数>,<参数>) 本函数的具体功能是：创建一个新线程并立即返回，返回值是所创建的新线程的标识号（如 果需要可以赋值给一个变量）。新线程随之启动，所执行的任务就是<函数>的代码，它应该 在程序中定义。调用<函数>时传递的实参由<参数>指定，<参数>是一个元组，如果<函数> 没有形参，则<参数>为空元组。当<函数>执行完毕返回时，线程即告结束。

下面用一个例子来说明 thread 模块的用法。程序 9.3 采用孙悟空拔毫毛变出小猴子的故事来演示主线程与它所创建的子线程的关系。主线程是孙悟空，子线程是小猴子。

【程序 9.3】eg9_3.py

```
# -*- coding: cp936 -*- import thread
def task(tName,n):
    for i in range(n):
        print "%s:%d\n" % (tName,i)
        print "%s:任务完成!回真身去喽...\n" % tName
        thread.interrupt_main()
print "<老孙>:我是孙悟空!"
print "<老孙>:我拔根毫毛变个小猴儿~~~"
thread.start_new_thread(task,("<小猴>",3))
print "<老孙>:我睡会儿,小猴儿干完活再叫醒我~~~\n"
while True:
    print "<老孙>:Zzzzzzz\n"
```

程序执行后，孙悟空（主线程）先说了两句话，然后创建小猴子，最后进入一个无穷循 环。小猴子创建后就立即启动，执行函数 task，该函数的任务只是显示简单的信息。task 函 数的最后一行调用 thread 模块中定义的 interrupt_main 函数，该函数的功能是在主线程中引 发 KeyboardInterrupt 异常，从而中断主线程的执行。如果没有这条语句，主线程将一直处于 无穷循环之中而无法结束。下面是程序的执行效果：

```
<老孙>:我是孙悟空!
<老孙>:我拔根毫毛变个小猴儿~~~
<老孙>:我睡会儿,小猴儿干完活再叫醒我~~~
<小猴>:0
<老孙>:Zzzzzzz
<小猴>:1
<老孙>:Zzzzzzz
<小猴>:2
<老孙>:Zzzzzzz
<小猴>:任务完成!回真身去喽...
<老孙>:Zzzzzzz
Traceback (most recent call last): File "eg9_3.py", line 15, in <module>
print "<老孙>:Zzzzzzz\n"
KeyboardInterrupt
```

从输出结果可见，主线程和子线程确实是在以交叉方式并行执行。 顺便说一下，由于程序 9.3 中使用了汉字，所以要在程序的第一行使用

```
# -*- coding: cp936 -*-
```

其作用是告诉 Python 解释器代码中使用了 cp936 编码的字符（即汉字）。 下面再看一个例子。程序 9.4 的主线程创建了两个子线程，两个子线程都执行同一个函数 task，但以不同的节奏来显示信息（每次循环中利用 sleep 分别休眠 2 秒和 4 秒）。注意， 与程序 9.3 不同，主线程创建完子线程后就结束了，留下两个子线程继续执行①。

【程序 9.4】eg9_4.py

```
# -*- coding: cp936 -*- import thread
import time
def task(tName,n,delay):
    for i in range(n):
        time.sleep(delay)
        print "%s:%d\n" % (tName,i)
print "<老孙>:我是孙悟空!"
print "<老孙>:我拔根毫毛变个小猴儿<哼>"
thread.start_new_thread(task,("<哼>",5,2))
print "<老孙>:我再拔根毫毛变个小猴儿<哈>"
thread.start_new_thread(task,("<哈>",5,4))
print "<老孙>:不管你们喽,俺老孙去也~~~\n"
```

下面是程序 9.4 的一次执行结果：

```
<老孙>:我是孙悟空!
<老孙>:我拔根毫毛变个小猴儿<哼>
<老孙>:我再拔根毫毛变个小猴儿<哈>
<老孙>:不管你们喽,俺老孙去也~~~
>>> <哼>:0
<哈>:0
<哼>:1
<哼>:2
<哈>:1
<哼>:3
<哼>:4
![](img/程序设计思想与方法293617.png)① 在作者所用的计算机平台（Windows XP + Python 2.7）上，主线
<哈>:2
<哈>:3
<哈>:4
KeyboardInterrupt
>>>
```

注意在“<哼>:0”之前的 Python 解释器提示符“>>>”，它表明主线程已经结束，控制 已返回给 Python 解释器。但后续的输出表明两个子线程还在继续执行。读者可以分析一下 输出结果所显示的<哼>、<哈>交叉执行的情况，并想想为什么是这样的结果。另外要注意， 由于主线程先于两个子线程结束，导致两个子线程执行结束后无法返回，这时可以用组合键 Ctrl-C 来强行中止子线程，屏幕上出现的 KeyboardInterrupt 就是因此而来。

threading 模块的用法

虽然 thread 模块用起来很方便，但它的功能有限，不如较新的 threading 模块。threading 模块采用面向对象方式来实现对线程的支持，其中定义了类 Thread，这个类封装了有关线 程创建、启动等功能。

Thread 类的使用有两种方式。第一种用法是：直接利用 Thread 类来创建线程对象，并 在创建时向 Thread 构造器传递线程将要执行的函数；创建后通过调用线程对象的 start()方法 来启动线程，以执行指定的任务。这是使用 threading 模块来创建线程的最简单方式。具体 语法如下：

```
t = Thread(target=&lt;函数&gt;,args=&lt;参数&gt;)
t.start()
```

可见，创建线程对象时，需向 Thread 类的构造器传递两个参数：线程所执行的<函数>以及该函数所需的<参数>。注意这里采用了关键字参数的传递方式。

下面的程序 9.5 与程序 9.4 的几乎是一样的，只是采用了 Thread 类来创建线程对象。

【程序 9.5】eg9_5.py

```
# -*- coding: cp936 -*-
from threading import Thread from time import sleep
def task(tName,n,delay):
    for i in range(n):
        sleep(delay)
        print "%s:%d\n" % (tName,i)
print "<老孙>:我是孙悟空!"
print "<老孙>:我拔根毫毛变个小猴儿<哼>"
t1 = Thread(target=task,args=("<哼>",5,2))
print "<老孙>:我再拔根毫毛变个小猴儿<哈>"
t2 = Thread(target=task,args=("<哈>",5,4))
t1.start()
t2.start()
print "<老孙>:不管你们喽,俺老孙去也~~~\n"
```

另一种使用 Thread 类的方法用到了线程对象的这么一个特性：当用 start()方法启动线程 对象时，系统会自动调用 run()方法。因此，只要我们将线程需要执行的任务代码写到 run() 方法中，就能在创建并启动线程对象后自动执行该任务。而将自己的代码写到 run()方法中 可以通过定义子类并重定义 run()的方式来做到。

总之，我们可以通过下列步骤来创建并启动线程，执行指定的任务。

（1）定义 Thread 类的子类。这相当于定制我们自己的线程对象。

（2）重定义 init**()**方法，即定制我们自己的构造器来初始化线程对象，例如可以添 加更多的参数。注意，定制构造器时首先应当执行基类的构造器 **Thread.**init ()，因为我 们定制的线程是原始线程的特例，首先要符合原始线程的要求。

（3）重定义 run()方法，即指定我们定制的线程将执行的代码。

（4）利用自定义的线程类创建线程实例，并通过调用该实例的 start()方法（间接调用 run()方法）或直接调用 run()方法来启动新线程执行任务。

程序 9.6 是采用上述方法的一个例子。

```
# -*- coding: cp936 -*-
from threading import Thread
from time import sleep
exitFlag = False
class myThread(Thread):
    def __init__ (self,tName,n,delay):
        Thread. __init__ (self)
        self.name = tName
        self.loopnum = n
        self.delay = delay
    def run(self):
        print self.name + ": 上场...\n" task(self.name,self.loopnum,self.delay)
        print self.name + ": 退场...\n"
    def task(tName,n,delay): for i in range(n):
        if exitFlag:
        print "<哼>已退场,<哈>也提前结束吧~~~\n" return
        sleep(delay)
        print "%s:%d\n" % (tName,i)
print "<老孙>:我是孙悟空!"
print "<老孙>:我拔根毫毛变个小猴儿<哼>"
t1 = myThread("<哼>",5,2)
t1.start()
print "<老孙>:我再拔根毫毛变个小猴儿<哈>\n"
t2 = myThread("<哈>",10,4)
t2.start()
while t2.isAlive():
    if not t1.isAlive():
        exitFlag = True
print "<老孙>:小猴儿们都回了,俺老孙去也~~~"
```

当线程启动后，就处于“活着（alive）”的状态，直到线程的 run()方法执行结束（不管 是正常结束还是因为发生异常而终止），该线程才结束“活着”状态。线程对象的 is_alive() 方法可用来检查线程是否活着。程序 9.6 中，主线程在创建并启动两个子线程 t1 和 t2 之后， 就一直检测 t2 是否还活着，如果 t2 活着就接着检测 t1 是否活着。当 t2 活着而 t1 已经结束， 则将一个用作退出标志的全局变量 exitFlag 设置为 True（初始值为 False）。而 t2 执行任务循 环时会检测 exitFlag 变量，一旦发现它变成了 True，就知道另一个线程已经结束，于是不再 执行后面的任务，直接结束返回。读者应该注意到这件事的意义，它意味着多个线程之间是 可以进行协作、同步的，而不是各自只管闷着头做自己的事情。

以下是程序 9.6 的一次执行结果：

```
<老孙>:我是孙悟空!
<老孙>:我拔根毫毛变个小猴儿<哼>
<哼>: 上场...
<老孙>:我再拔根毫毛变个小猴儿<哈>
<哈>: 上场...
<哼>:0
<哼>:1
<哈>:0
<哼>:2
<哈>:1
<哼>:3
<哼>:4
<哼>: 退场...
<哈>:2
<哼>已退场,<哈>也提前结束吧~~~
<哈>: 退场...
<老孙>:小猴儿都回了,俺老孙去也~~~
>>>
```

线程有名字，可通过 getName()和 setName()方法读出或设置线程名。总是有一个主线程对象，它对应于程序的初始控制流。

并发计算中的同步问题

多个线程之间如果只是彼此独立地执行各自的任务，事情就简单了。但是实际应用中常常需要多个线程之间进行合作，合作的线程往往要存取一些公共数据。由于多个线程的执行 顺序是不可预测的，这就有可能导致公共数据处于不一致的状态。因此，如果允许多个线程 并发读写公共数据，就必须对多线程的交互进行控制，以保护公共数据的正确性。

程序 9.6 演示了两个线程通过一个全局变量进行协同的例子。

又如，Thread 对象具有一个 join()方法，一个线程对象 t1 可以调用另一个线程对象 t2 的 join()方法，这导致 t1 暂停执行，直至 t2 执行结束（或者执行一个指定的时间）。可见，join 方法能实现让一个线程等待另一个线程以便进行某种同步的目的。

threading 模块还实现了更一般的同步机制，在此就不介绍了。

9.3.5 小结

多线程编程属于比较复杂的程序设计任务，即使对专家也不是容易的事情。这是因为多 线程在执行上具有不确定性，线程一旦启动，他们之间的相互依赖和相互作用的结果就是不 可预测的。《西游记》中的这段描写或许能帮助读者想象多线程并发执行的复杂性：

悟空见他凶猛，即使身外身法，拔一把毫毛，丢在口中嚼碎，望空喷去，叫一声“变”！即变做三二百个小猴……把个魔王围绕，抱的抱，扯的扯，钻裆的钻裆，扳脚的扳脚， 踢打挦毛，抠眼睛，捻鼻子，抬鼓弄，直打做一个攒盘。

不过，有很多看似需要并发计算的问题实际上可以用事件驱动（参见第 8 章）而不是用 线程来编程。事件驱动在多数情况下更容易实现，因为事件驱动只有一个执行流，没有并发 及同步问题。涉及 GUI 的问题就可以用事件驱动来解决，而科学计算的并行化则适合用多 线程来解决。总之，应当只在真的需要并发计算时才使用线程。

9.4 练习

1. 利用 random 或 randrange 函数来实现下列计算任务。

（1）表示打靶所得环数（0～10 环）的整型随机数；

（2）区间[-0.5,0.5]内的浮点型随机数；

（3）表示投掷一个骰子所得结果的随机数；

（4）表示投掷两个骰子所得结果的随机数。

2. 修改程序 9.2，以 A、B 两球员各自的发球得分概率表示技术水平，并将发球、换发球添加到程序中。

3. 设计并实现模拟排球比赛的程序，以研究只有发球方才能得分的老规则与每回合得分的 新规则对胜负是否有影响。如果有，新规则有利于强队还是弱队。

4. 编程模拟骰子游戏：玩家首先投掷两个骰子，如果掷出的是 2、3 或 12 点，则玩家输；

如果掷出的是 7 或 11 点，则玩家赢；如果是其他点数（设为 p 点），则继续投掷，一直到掷出 7 点或者再次掷出 p 点为止，掷出 7 点则玩家输，再次掷出 p 点则玩家赢。模拟多局，并估算玩家的获胜概率。

5. 蒙特卡洛技术可用于估计p的值。假设有一块圆形飞镖板，正好嵌在一个正方形橱柜里。现在来随机投掷飞镖，命中飞镖板的次数与命中橱柜（即飞镖板没有覆盖的四个角落）的次数之比，是由飞镖板与橱柜面积决定的。设 n 是总的投掷次数（落在橱柜范围内），h 是命中飞镖板的次数，则p≈4h/n。编程模拟飞镖游戏，输入 n，输出p的估计值。提示：如果正 方形以原点为中心且大小为 2x2，则可以用 2*random() - 1 来生成落在正方形中的随机点的 x 和 y 坐标。如果 x2+y2≤1，则该点落在飞镖板内部。

6. 编程模拟一手掷 5 个骰子，估计 5 个骰子点数全部相同的概率。

7. 模拟一维随机漫步（random walk）：在一条很长的笔直马路上漫步，步行方向由掷硬币决定。即如果掷出正面，则向前走一步；否则向后走一步。记录走 n 步后离出发点的距离 d。重复多次试验，看看 d 的平均值等于多少。

8. 模拟二维随机漫步：在一个平面上，每次随机向前、向后、向左或向右走一步。如果走 n 步，离出发点距离 d 是多少？需要重复多次求得 d 的平均值。

9. 修改第 8 题，改成每一步允许向任何方向迈步。提示：利用 angle = random() * 2p随机生成一个方向角（前进方向与 x 轴的夹角），走到由 x = x + cos(angle)和 y = y + sin(angle)决定 的位置。作图显示漫游轨迹。

10. 编写一个程序，其中创建 2 个线程，一个线程用来计算 2～10000000 之间的素数个数，另一个线程用来计算 10000000～20000000 之间的素数个数。哪个范围内的素数多？

第 10 章 算法设计和分析

利用计算机解决问题的关键是设计出合适的算法。对特定问题设计出求解算法，体现了 程序设计这种智力活动的创造性的一面。从事创造性活动需要创造性思维，而不能仅仅依靠 机械的模仿。虽然算法设计并没有一定之规，但计算机科学家总结出了一些行之有效的设计 方法，掌握这些方法对于利用计算机解决问题具有重要意义。利用计算机解决问题，并非只 要设计出正确的算法就行了，还需要分析算法的复杂度。本章主要介绍一些常用的算法设计 方法，以及对算法时间复杂度的分析。

10.1 枚举法

问题求解中常用的一种算法设计方法是枚举策略。给定问题 P，如果知道 P 的可能解构 成一个有限集合 S = {s1, s2, ..., sn}，则可以逐一列举每个 si，检验它是否确实是 P 的解，这 就是枚举法。枚举法简单而直接，算法容易设计实现，但当可能解集合 S 很大时，枚举策 略的效率很差。实际使用枚举法时，经常利用各种已知条件来从 S 中排除掉一部分不可能 情形，从而优化枚举过程。下面通过几个例子来说明枚举策略在设计算法中的应用。

线性搜索

首先看一个程序设计中常见的问题——搜索（或称查找）问题：给定数据集合 D，在 D 中查找指定数据 x。

搜索问题看上去很容易解决，一个显而易见的做法是：反复从 D 中读取下一个数据， 看看它是否 x，搜索结果是要么找到 x，要么发现 D 中没有 x。然而，这个“算法”是有问 题的，因为它需要一个关键操作——“读取取下一个数据”，而“下一个”未必是良定义的。 打个比方，如果一群人站成一排，当我们要从中找出张三时，可以采取按排队次序逐个询问 的策略。但如果这群人散乱无规则地站在一起，我们该如何循着一个有条理的过程找出张三 来呢？如何决定“下一个”要询问的人？

可见，要想在一个数据集合中找到指定数据，就必须能够按某种系统化的方式逐个列举 集合元素，并与指定数据进行比较。这就是枚举策略在搜索问题中的应用。

如果将大量数据存储在一个列表中，则使用枚举策略很合适，因为列表是通过位置索引 来访问其中数据成员的，“读取下一个数据”是良定义的操作，只要将当前位置索引加 1 即 可得下一个数据的索引。下面定义的函数 find()实现了这种搜索策略：给定数据列表 list 和 需要查找的数据 x，逐个取出 list 的成员并与 x 进行比较。如果某个成员就是 x，则返回该 成员在列表中的位置索引；如果 list 中没有 x 则返回-1。

```
>>> def find(list,x):
for i in range(len(list)): if list[i] == x:
return i return -1
>>> find([2,4,6,8],6) 2
>>> find([2,4,6,8],7)
-1
```

find()函数对列表 list 从头到尾进行扫描，扫描过程中检验每一个成员是否 x，这个算法 称为线性搜索（linear search）算法。线性搜索算法很容易设计实现，而且当数据量不太大时，算法的性能也还可以。更重要的是，由于线性搜索是枚举每一个数据成员，因此适用于 无序数据集合，即数据没有按特定的大小次序排列。

然而，当数据量很大时，逐个枚举集合中的数据就变得非常低效。这时只能通过更好地 组织数据，利用额外信息来提高搜索效率，尽量避免逐个检查所有数据。例如，假设列表数 据从小到大有序排列，那么在枚举过程中一旦发现当前取出的数据大于 x，就不必再继续搜 索

了，可以直接下结论说找不到 x。这种改进可以提高线性搜索算法的性能，但改善得很有限。事实上，在数据有序的情况下，存在比线性搜索算法好得多的算法（见 10.2）。

线性搜索算法只适用于“一维”搜索空间，即所有数据排列成一排的情形。考虑在如下 矩阵中查找某个数据的问题：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

这时显然无法直接采用线性搜索算法。在类似矩阵这样的“二维”搜索空间中，如何枚举每 一个数据呢？这个问题其实在第 3 章中介绍循环语句时就讨论过，为了遍历（即枚举）这样 的二维空间，可以采用嵌套的循环语句。例如下面这个 find2D()函数实现了在 row 行、col 列的矩阵 matrix 中查找数据 x 的枚举算法：

```
>>> def find2D(matrix,row,col,x):
    for i in range(row):
        for j in range(col):
            if matrix[i][j] == x:
                return (i+1,j+1)
        return -1
>>> find2D([[1,2,3],[4,5,6]],2,3,6) (2, 3)
>>> find2D([[1,2,3],[4,5,6]],2,3,7)
-1
```

显然，这个做法可以扩展到更多维的搜索空间，利用 n 层嵌套循环即可枚举 n 维搜索空 间中的数据。

求解不定方程

有时问题的所有可能解并没有像上例那样明确地存储在某个具体集合（如列表）中，而 是构成一个无形的搜索空间，那该如何枚举可能解呢？这需要具体问题具体分析，根据问题 的特点设计枚举方式。下面是一个典型的例子。

中国古代数学著作中有一道“百钱买百鸡”问题：假设公鸡每只 5 元钱，母鸡每只 3

元钱，鸡雏每三只 1 元钱，用一百元钱买了一百只鸡，问公鸡、母鸡和鸡雏各买了几只？具备初等代数知识的人都不难列出如下方程组来求解这个问题：

```
x + y + z = 100
5x + 3y + z/3 = 100
```

其中 x、y、z 分别表示公鸡、母鸡和鸡雏的个数。 此方程组有三个未知数却只有两个方程式，属于数学中所称的不定方程。人工求解不定

方程通常会利用方程变形、未知数代换以及分析各种约束条件等技巧，而绝不会采用枚举所有可能解进行检验的方法，因为可能解构成的空间通常非常庞大。然而，计算机的优点恰恰在于能够高速地、机械地执行大量的检验任务，因此采用枚举策略来解不定方程是简单而直

接的做法。问题是如何枚举各种可能解呢？对于百钱买百鸡问题，显然只需为三个未知数做各种可能的赋值，然后检查是否满足上述两个方程式即可。各未知数的可能值都在 100 之内（因为只买了 100 只鸡），所以利用枚举法很容易得到下列程序：

```
>>> for x in range(100):
        for y in range(100):
            for z in range(100):
                t = x + y + z
        m = 5*x + 3*y + z/3
        if t == 100 and m == 100:
            print "x=",x,",y=",y,",z=",z
x=  0  ,y=  25  ,z=  75
x=  3  ,y=  20  ,z=  77
x=  4  ,y=  18  ,z=  78
x=  7  ,y=  13  ,z=  80
x=  8  ,y=  11  ,z=  81
x=  11  ,y=  6  ,z=  83
x=  12  ,y=  4  ,z=  84
```

采用枚举策略时应当尽量减小可能解集合，以便提高枚举效率。上面这个程序的效率显 然太差，因为三重嵌套循环实际上要枚举 100×100×100 种 x、y、z 组合。其实稍加思考就 能找到减小需要检验的可能解的数目的方法。首先，不需要三层嵌套循环，因为当 x 和 y 的值给定，z 的值就确定了（即 100–x–y），没有必要再去枚举 z；其次，x 的可能值不超 过 20（否则钱不够），同理 y 的可能值不超过 33；最后，依题意每种鸡应当都至少买 1 只，没有必要考虑等于 0 的情形。将这些分析落实到编程中，即可得效率更高的代码：

```
>>> for x in range(1,20):
        for y in range(1,33):
            z = 100 - x - y
            m = 5*x + 3*y + z/3
            if m == 100:
                print "x=",x,",y=",y,",z=",z
x=  3  ,y=  20  ,z=  77
x=  4  ,y=  18  ,z=  78
x=  7  ,y=  13  ,z=  80
x=  8  ,y=  11  ,z=  81
x=  11 ,y=  6   ,z=  83
x=  12 ,y=  4   ,z=  84
```

利用问题中的各种约束条件往往可以减少搜索空间或者优化枚举过程。例如，假设为 “百钱买百鸡”问题附加一个条件“尽量多买公鸡”，那么可以这样优化算法：最外层对 x 的循环中改用 range(20,0,-1)，以便尽快找到满足条件的值，得到第一个解之后就可以 终止程序，不必再找其他解了。

通过以上例子，我们看到枚举算法的核心思想是对问题的每一个可能解进行检验，看看 是否满足特定条件，这个枚举过程在编程时是通过循环语句和条件语句实现的。对于一些复 杂问题，如果嵌套循环的层数不确定或者层数太多，直接使用循环语句和条件语句实现枚举 检验是不合适甚至不可能的，这时可以考虑采用递归技术（见 10.2）。

当问题规模较大时，可能解的空间也很大，采用枚举策略会导致效率很差。但是，鉴于 枚举算法设计简单，调试也容易，对于规模较小的问题是很好的策略。即使对于大规模的复 杂问题，枚举策略也可以作为整体求解算法的子算法出现。

最后总结一下采用枚举策略设计算法的一般步骤：

(1) 确定枚举对象、枚举范围和判定条件；

(2) 枚举各可能解，逐一验证是否所需的问题解。

(3) 尽量减小枚举范围，提高算法效率。

10.2 递归

我们已经知道，循环是必不可少的基本流程控制结构之一，在编程中时时会用到循环语 句。但出乎意外的是，一个编程语言实际上可以不提供循环语句①！因为有另一种语言构造 可以替代循环，这就是递归。

读者也许听说过“循环定义”，即在定义概念 A 的时候直接或间接地用到了 A 自身。例 如将“逻辑学”定义成“研究逻辑的科学”，这实际上是同语反复，并未揭示任何新的内涵；又如将“美丽”定义成“漂亮”，再将“漂亮”定义成“美丽”，这种循环定义实际上也是同 语反复。循环定义是一种常见的逻辑错误，应尽量避免使用，但在数学和程序设计中，我们 经常在一个函数的定义中直接或间接地用到该函数自身，这种函数称为递归（recursive②） 函数。通过下面的讨论我们会看到，这种递归定义不同于循环定义，它能够明确地定义出函 数的意义。

递归是一种强大的算法设计思想和方法，利用递归可以轻松解决很多难题。下面我们通 过例子来介绍这种方法。

阶乘

数学中的阶乘运算通常用下式定义：

```
n! = n x (n - 1) x (n - 2) x ... x 2 x 1
```

注意，当 n 为 0 时，其阶乘被定义为 1。

如果要编程计算 n 的阶乘，可以采用以前介绍过的累积算法模式来实现，累积算法的关 键部分是一个循环语句。下面是此方法的实现代码及执行实例：

> > > def fac(n): if n == 0: return 1 else: f = 1 for i in range(1,n+1): f = i * f return f ```
>
> ① 有一类函数式编程语言（如 Scheme）就不提供循环语句构造。
>
> ② 英文 recur 的原意为再次发生、重新出现等。被定义的术语又出现在定义之中，就是递归。

```
>>> fac(4)
24
>>> fac(40)
815915283247897734345611269596115894272000000000L
```

下面我们用另一种方式来观察阶乘的定义。在阶乘定义式中，等号右边的第一个 n 之后 的部分是什么？稍加思考即可看出就是(n-1)的阶乘，即阶乘定义式可写成：

```
n! = n x (n - 1)!
```

这个式子的含义是：n 的阶乘定义为 n 乘(n-1)的阶乘。我们看到，“阶乘”的定义中用到了 “阶乘”本身，这就是递归定义。

现代编程语言都支持递归函数，Python 也不例外。读者也许会将上面的递归定义式直 接翻译成如下 Python 函数：

```
def fac(n):
    return n * fac(n-1)
```

但这个定义是错误的。如果执行这个函数，将会形成如下调用链条：

```
fac(n) => fac(n-1) => ... => fac(1) => fac(0) => fac(-1) => fac(-2) => ...
```

显然，递归将无穷无尽地延续下去①。

有效递归定义的关键是具有终止条件，使得到达某一点后不再递归，否则会导致像无穷循环一样的后果。对阶乘来说，当 n=0 时，n!的值定义为 1，此时无需递归。在上面的 fac 函数中添加这个终止条件，即可得正确的递归版阶乘函数：

```
>>> def fac(n):
        if n == 0:
            return 1
        else:
            return n * fac(n-1)
>>> fac(4)
24
>>> fac(40)
815915283247897734345611269596115894272000000000L
```

为了理解递归函数的执行过程，需要回顾第 4 章中介绍的函数调用与返回的知识。图10.1 展示了 fac(2)的计算过程。

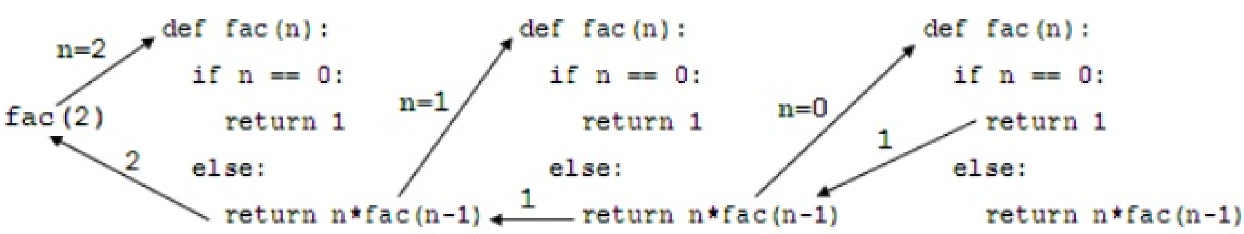


图 10.1 fac(2)的计算过程

> ① 事实上编程语言中的递归层数是有限制的，当突破限制时递归过程会终止。

计算 fac(n)时，由于每次递归都导致计算更小的数的阶乘，因此这个递归过程迟早会到 达计算 fac(0)的情形。而根据 fac()的定义，fac(0)直接返回 1，无需递归计算。我们称这种情 形为递归定义的奠基情形。对于奠基情形，递归函数能够直接计算结果。

要说明的是，上面的阶乘函数定义其实仍然有 bug：当 n 的初始值小于 1 时，调用 fac(n) 会导致无穷递归！解决这个问题很容易，只需在程序开始处检查 n 是否为负数即可，并且仅 当 n 为非负自然数时才能计算阶乘。编写递归程序时很容易在终止条件上面犯错误，作为好 的编程习惯，我们应当围绕递归奠基情形测试各种情形。

还要说明一点，每次递归调用 fac()都相当于调用一个新的函数，系统将为该函数的局 部变量和参数分配新的空间，与其他 fac()调用的局部变量和参数完全没有关系。初学者在 这一点上也会经常犯错误，以为各递归调用中使用的变量是全局共享的。在图 10.1 中有三 次对 fac(n)的调用，这三次调用应当视为独立的三个函数，其中用到的参数 n 应当视为三个 相互独立的局部变量。

列表处理

递归对于处理列表是非常有用的，因为列表本身就是“递归结构”——任一列表都可看 作是由第一个数据成员与剩余数据列表组成的，即：[a1,a2,...,an]可视为由 a1 和[a2,...,an]组成。 编程处理这个列表时，只需要单独考虑如何处理 a1，而对[a2,...,an]的处理可以通过递 归调用来解决。显然，每次递归都导致处理一个更短的列表，如此递归下去终将到达空列表 情形，这正可作为奠基情形。在 Python 中通过索引很容易取得列表 list 的第一个数据和剩 余数据列表，它们分别是 list[0]和 list[1:]。

作为例子，下面我们写一个递归函数来逆向显示列表的数据，即将[a1,a2,...,an]显 示为

```
an,...,a2,a1
```

根据列表的“递归结构”性质，不难形成这样的递归思考：为了逆向显示 list，只需先 逆向显示 list[1:]，然后显示 list[0]；当剩余数据列表为空时停止递归。这个递归思考可以直 接翻译成如下 Python 代码：

```
>>> def revList(list):
        if list != []:
            revList(list[1:])
            print list[0],
    else:
        return
>>> revList([1,2,3,4,5])
5 4 3 2 1
```

对于简单列表的处理任务，用 for 循环语句也很容易实现；但当列表很复杂（例如列表 中的元素本身可能是列表），用循环语句就很难编程，而用递归则可以很容易地解决问题。 作为练习，读者不妨思考一下如何逆向显示如下形状的列表：

```
[1,[2,3],4,[5,6,[7,8],9]]
```

二分搜索

10.1 节中介绍了线性搜索算法，读者已经知道线性搜索的优点是适合无序的数据列表，缺点是不适合大量数据。当列表中的数据有序时，存在更好的搜索策略，这个策略的基本思 想可以通过一个猜数游戏展现出来。

假设某甲心中想好了一个 100 以内的自然数，让某乙来猜这个数。某乙每猜一次，某甲 都会告诉他猜对了、猜大了或猜小了三种情形之一。某乙该采用什么策略来玩这个游戏呢？某乙可以每次都随机猜一个数，也可以系统化地按 1、2、3、……的顺序猜（此即线性搜索），但这两种策略平均需要猜很多次才能猜中。最好的策略是先猜 1～100 的中间数 50，如果猜中自不必说，如果猜大了则接下去猜 1～49 的中间数 25，如果猜小了则接下去猜 51～100 的中间数 75。依此类推，每次都猜可能值范围的中间值，直至猜中。这个策略就是我们要 介绍的二分搜索（binary search）算法。

下面我们利用二分搜索来解决在一个有序数据列表 list 中查找指定数据 x 的问题。先看 如何利用循环来实现二分搜索。算法的核心是一个循环，每一次循环都检查当前搜索范围的 中间数据是否等于 x；不等的话，根据大小信息重新设定搜索范围；如果找到了 x，或者没 有剩余数据了，则循环终止。为了便于设定搜索范围，可以用两个变量 low 和 high 分别记 录搜索范围的两端，每次循环后根据比较结果调整这两个变量即可重新设定搜索范围。代码 如下：

```
def binary(list,x): low = 0
    high = len(list) - 1
    while low <= high:
        mid = (low + high) / 2
        if list[mid] == x:
            return mid
        elif list[mid] > x:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

再看二分搜索的递归实现。二分搜索算法可以这样表达：检查当前搜索范围的中间数据， 如果该数据就是目标数据，则算法结束；如果不是，则选择某一半范围重新进行二分搜索。 这段话可以翻译成如下伪代码：

```
二分搜索算法：在范围 list[low]到 list[high]之间查找 x
    取当前范围的中间数据 m；
    如果 m 等于 x 或者 m 不存在则算法结束；
    如果 x &lt; m 则在范围 list[low]到 list[mid-1]之间查找 x，
    否则在范围 list[mid+1]到 list[high]之间查找 x。
```

这个算法中有三处（见划线部分）涉及几乎相同的操作，这正是二分搜索的递归性质的 体现。奠基情形是找到了目标值或者检查完所有数据都未找到目标值，这时将不再递归。由 于每次递归调用都将搜索空间减小了一半，因此迟早会到达奠基情形。下面给出递归版本二 分搜索的 Python 代码实现。注意与循环版本不同的是，每次递归都需要指明搜索范围，因 此我们将搜索范围的两个端点 low 和 high 也作为函数参数。

```
>>> def recBinSearch(list,x,low,high):
    if low > high:
        return -1
    mid = (low + high) / 2
    m = list[mid]
    if m == x:
        return mid
    elif x < m:
        return recBinSearch(list,x,low,mid-1)
    else:
        return recBinSearch(list,x,mid+1,high)
>>> recBinSearch([1,3,5,7,9],5,0,4) 2
>>> recBinSearch([1,3,5,7,9],6,0,4)
-1
```

阶乘和二分搜索这两个例子说明，许多问题既可用循环（或称迭代）来实现，也可用递 归来实现。很多情况下，两种方法在设计上都很容易；但对有些问题，迭代算法很难设计， 而递归算法则非常容易得到，例如下面的 Hanoi 塔问题。

Hanoi 塔问题

Hanoi 塔问题是体现递归方法强大能力的经典例子，该问题涉及如下故事：在某个寺庙 里有三根柱子（不妨称为 A、B、C 柱），A 柱上有 n 个同心圆盘，圆盘尺寸各不相同，并且 小盘叠在大盘之上，B 柱和 C 柱为空（参见图 10.2）。寺庙的僧侣们有一项任务：将 n 个圆 盘从 A 柱移到 C 柱，移动过程中可以利用 B 柱作为临时存放柱。具体的移动圆盘的规则是：

- 圆盘只能放在柱子上；
- 每次只能移动位于任一柱子顶部的圆盘；
- 大圆盘永远不能置于小圆盘之上。

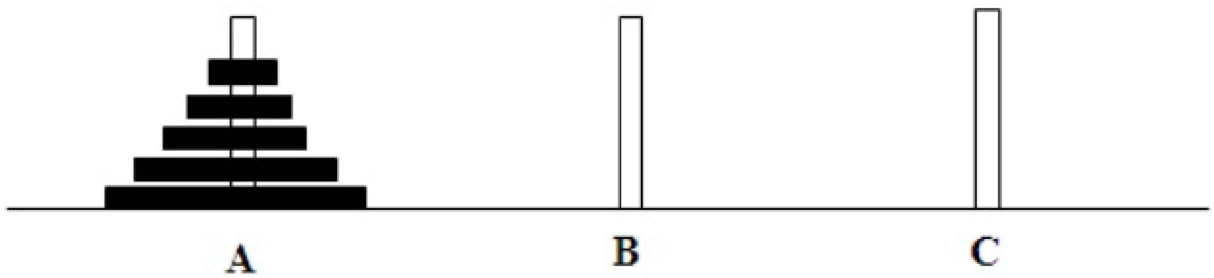


图 10.2 Hanoi 塔问题

下面我们来设计解决此问题的算法，该算法能够给出搬运步骤。例如对于 n = 3 的情形，算法将显示如下移动过程（其中 A -> C 表示将 A 柱顶部圆盘移至 C 柱顶部，余类推）：

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Hanoi 塔问题看上去有点难度，但如果采用递归方法，算法是非常简单的。稍加思考即 可明白，为了将 A 柱上的所有圆盘移到 C 柱，必然需要有一步将底部的最大圆盘从 A 柱移 到 C 柱，而为此又必须先将最大圆盘上面的 n - 1 个圆盘从 A 柱移到 B 柱，从而形成最大 圆盘之上没有其他圆盘、同时 C 柱上也没有圆盘的局面（参见图 10.3）。

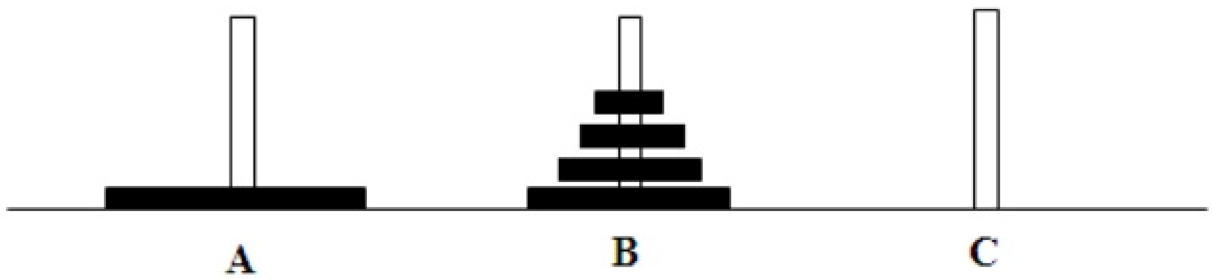


图 10.3 为移动最大圆盘必须形成的格局

至此，可以将最大圆盘从 A 柱移到 C 柱，显然接下去再也不需要移动这个圆盘了，因此可视为不存在。这时形成的局面（图 10.4）与初始局面非常相似，即：B 柱上有 n - 1 个 圆盘，A 柱和 C 柱为空（无视最大圆盘）。于是任务变成了：将 n - 1 个圆盘从 B 柱移动到 C 柱，移动过程中可以利用 A 柱作为临时存放柱。将这里的黑体部分文字与前面的初始问 题文字相比较，即可看出 Hanoi 塔问题的递归性质：一旦最大圆盘到达目的地，剩下来的问 题恰好又是初始 Hanoi 塔问题，只不过问题规模变成了 n - 1，并且 A 柱和 B 柱的角色相互 交换。

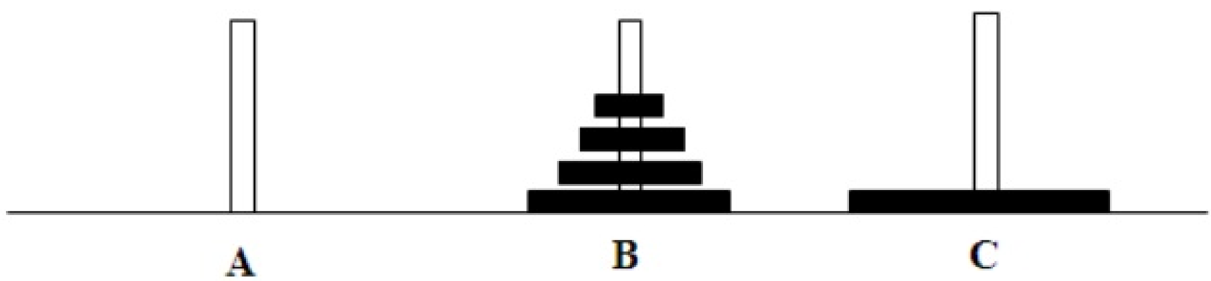


图 10.4 最大圆盘就位之后的格局

根据以上分析，容易得到解决 Hanoi 塔问题的算法。下面是算法的伪代码：

```
算法：将 n 个圆盘从 A 柱移到 C 柱，以 B 柱作为临时存放柱。
      将 n - 1 个圆盘从 A 柱移到 B 柱，以 C 柱作为临时存放柱；
      将最大圆盘从 A 柱移到 C 柱；
      将 n - 1 个圆盘从 B 柱移到 C 柱，以 A 柱作为临时存放柱。
```

从算法中可见，通过递归，我们将规模为 n 的问题转化成了两个规模为 n – 1 的问题。 如此递归下去，最终将转化成规模为 1 的问题。而 n = 1 的 Hanoi 塔问题是平凡的，直接移 动一步即可，不再需要递归，这就是奠基情形。有了奠基情形，每次递归又导致问题规模变 小，可见上述递归算法能正确终止。下面我们给出对上述算法的 Python 实现，并对 n = 3 的 情形进行验证。代码中 hanoi 函数的参数分别表示圆盘个数和三根柱子（源、目的地、临时 存放）。

```
>>> def hanoi(n,source,dest,temp):
    if n == 1:
        print source,"->",dest
    else:
        hanoi(n-1,source,temp,dest)
        hanoi(1,source,dest,temp)
        hanoi(n-1,temp,dest,source)
>>> hanoi(3,"A","C","B")
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

至此，一个看上去挺难的问题通过递归就轻松地解决了。读者有兴趣的话不妨试试如何 利用循环（迭代）来解决 Hanoi 塔问题。

最后对递归方法做个小结。

递归是非常重要的算法设计方法，在解决很多具有递归性质的问题、结构的时候，设计

递归算法往往是直接而简单的。递归定义必须满足以下条件才是良定义的：

- 有一个或多个无需递归的奠基情形；
- 递归总是针对规模更小的问题。

有了这两个条件，递归链最终将到达奠基情形，从而使递归过程终止。

虽然递归算法容易设计、实现，也容易理解，但递归是有代价的。由于递归涉及大量的 函数调用，因此需要耗费较多的内存和较长的执行时间，即递归算法的效率较差。而迭代算 法不涉及函数调用，故速度更快，更节省内存。

10.3 分治法

分治法（divide-and-conquer）是解决问题的一种常用策略，其思想是将难以处理的较大 问题分解为若干个较小的子问题，然后分别解决这些子问题，并从子问题的解构造出原问题 的解。“分”是指将原问题分解，“治”是指解决问题。

“分治”仅提及了分而治之的过程，而未提及此方法的另一个特点——递归。当我们将 大问题分解成子问题后，经常会发现子问题与大问题本质上是相同的问题，因此可以利用递 归方法来设计算法。所以，分治法常常与递归法结合起来使用。

下面我们通过排序问题来介绍分治法的应用。排序问题是指将一个数据集合中的所有数 据按从小到大的顺序（严格递增或非递减）重新排列①。计算机科学家发明了很多排序算法， 本节主要介绍利用分而治之思想设计的归并排序算法，但为了进行比较，我们先介绍没有什么“技术含量”的选择排序算法。

选择排序

选择排序是一种朴素的排序方法，普通人都很容易想到。其思想是：先从全体 n 个数据 中找出最小值，并将该最小值排在第一个位置；然后从剩下的 n-1 个数据中再次找出最小值，这个最小值实际上是全体数据的次小值，我们将它排在第二个位置；依此类推，直至从剩下 的 2 个数据中找出最小值，排在第 n-1 个位置，而剩下的最后一个数据（全体数据中的最大值）可以直接排在第 n 个位置。

> ① 当然也可以按从大到小的顺序（严格递减或非递增）排列，这在解决方法上并没有什么本质差别。

选择排序方法的关键步骤是找出当前剩余数据中的最小值。我们在 3.6 节中讨论过这个 问题①，并且设计了一个很好的算法：逐个检查每一个数据，并记录当前见到的最小值；当 数据检查完毕，所记录的数据就是全体数据中的最小值。下面我们利用这个求最小值的方法 来实现选择排序算法。

算法的核心部分是一个循环，每一轮循环找出剩余数据中的最小值，并将该值放到合适 位置。假设数据在列表 list 中，则第一次循环找出 list[0:n-1]中的最小值，并将该值存入 list[0]处（原来的 list[0]数据需要挪地方，见下面介绍的实现技巧）。第二次循环从 list[1:n-1]中找出最小值，并存入 list[1]处；依此类推，第 n-1 次循环将 list[n-2:n-1]中的最小值存入 list[n-2]， 而剩下的最后一个数据自然只能存入 list[n-1]。至此，list 中存储的数据即为从小到大有序 排列的。

实现此算法时，如果没有额外的存储空间，只使用 list 本身的空间来排序，则在第一次 循环中将最小值放入 list[0]时，原先存储在其中的数据就会被覆盖。为了保留这个数据，一 个简单的技巧是将 list[0]与找到的最小值交换。即，假如最小值是 list[k]，则执行

```
list[0],list[k] = list[k],list[0]
```

其他轮次的处理也是一样。为此，在循环中需要用一个变量记录最小值的位置索引。 下面的 Python 代码实现了以上设计思想，其中每轮循环找出 list[i:n-1]中的最小值（用变量 min 记录其索引位置），并放入 list[i]中。

```
>>> def selSort(list):
        n = len(list)
        for i in range(n-1):
            min = i
            for j in range(i+1,n):
                if list[j] < list[min]:
                    min = j
            list[i],list[min] = list[min],list[i]
>>> datalist = [5,2,8,3,4]
>>> selSort(datalist)
>>> print datalist
[2, 3, 4, 5, 8]
```

注意，与 3.6 中最小值算法不同的是，这里找最小值时并非记录最小值本身，而是记录最小值的索引位置 min，即 list[min]才是当前最小值，这是为了使列表数据交换位置更方便。另外，循环变量 i 只需从 0 取到 n-2，因为当前 n-1 个数据就位后，最后一个位置自然就是最 大值。

选择排序算法很容易设计实现，并且当数据量不大时效率也还可以，但当数据量很大时 性能很差。采用分治法可以设计一种更好的排序算法，即归并排序。

> ① 3.6 中讨论的是求最大值，但算法稍加改变即可用于求最小值。

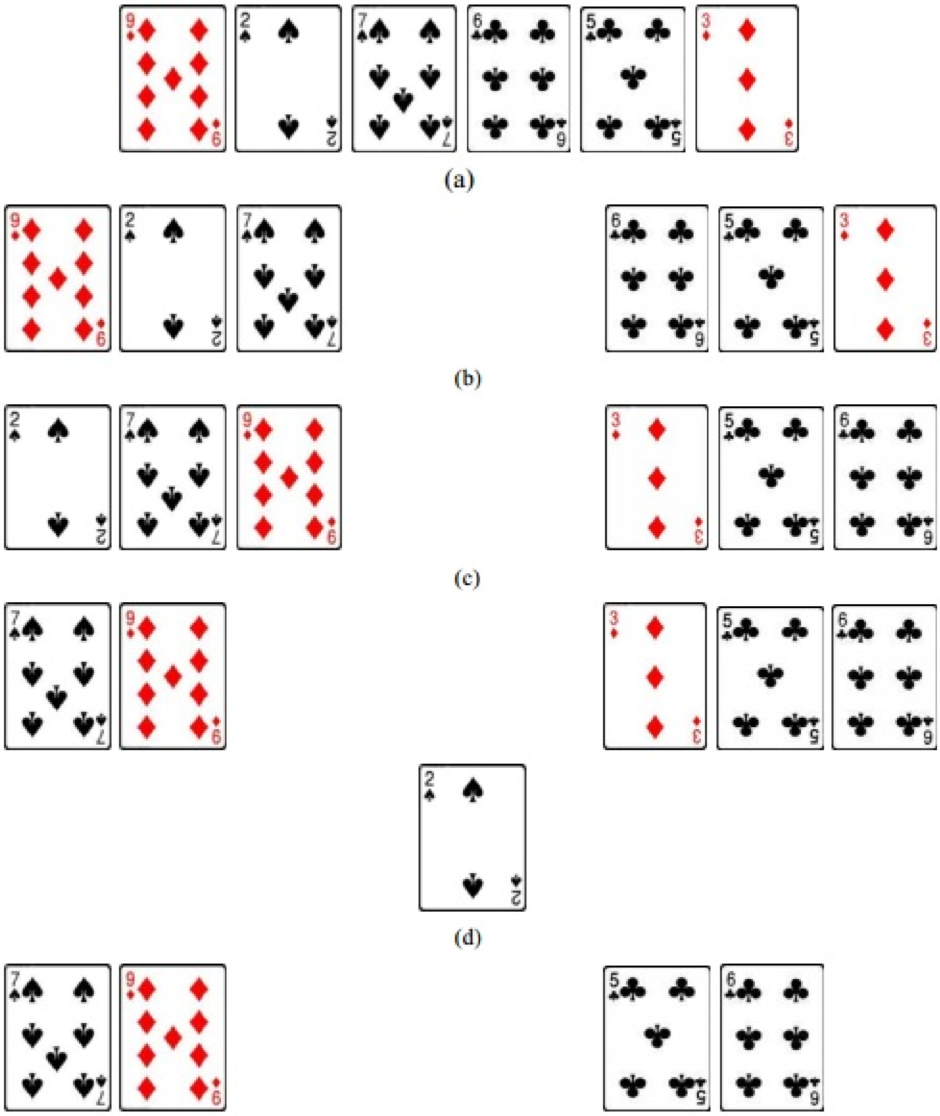
归并排序

人们在玩扑克牌的时候，经常将手上的牌排成特定的顺序，比如按花色或按大小排序。

如果分到的牌不多，玩家一般用一只手将牌呈扇形握持，另一只手去整理排序。然而，如果玩的是用到两三副牌的游戏，每个玩家分到的牌很多，那么玩家就会有手太小难以排序的烦恼。这时，如果旁边坐着个观战者，玩家可以请这个观战者帮着拿一些牌，两人分别将手中不多的牌进行排序，然后再合并两手牌以完成全部牌的排序。这就是归并排序的基本思想，它将大任务分解成较小任务，解决了较小任务后再合并结果。下面我们详细介绍这种利用分治法进行排序的方法。

给定一个较大的数据集合 S，先将数据平分为两部分 S1 和 S2，然后分别对 S1 和 S2 进行排序，从而得到两个“局部有序”的序列。接下去将这两个局部有序序列合并成为“全局有 序”序列，这个过程称为归并（merge）。假设用序列 S3 存储归并结果，则具体归并方法是： 第一轮，两个局部有序的序列 S1 和 S2 分别拿出自己的局部最小值进行比较，其中更小者显 然是全局最小值，因此应放入 S3 的第一个位置。如果全局最小值来自 S1，则 S1 中原来排在 该最小值后面的数据成为新的局部最小值。第二轮，再次比较 S1 和 S2 的局部最小值，其中 更小者实际上是全局第二小的数据，因此应放入 S3 的第二个位置。第三轮以下依此类推， 不

断比较 S1 和 S2 的局部最小值，并将更小者放入 S3，直至 S1（或 S2）的所有数据都已放入 S3。最后，只需将 S2（或 S1）的剩余数据按序放入 S3 的尾部，即可得到全局有序序列。图 10.5 用整理扑克牌的例子展示了这个归并排序过程。



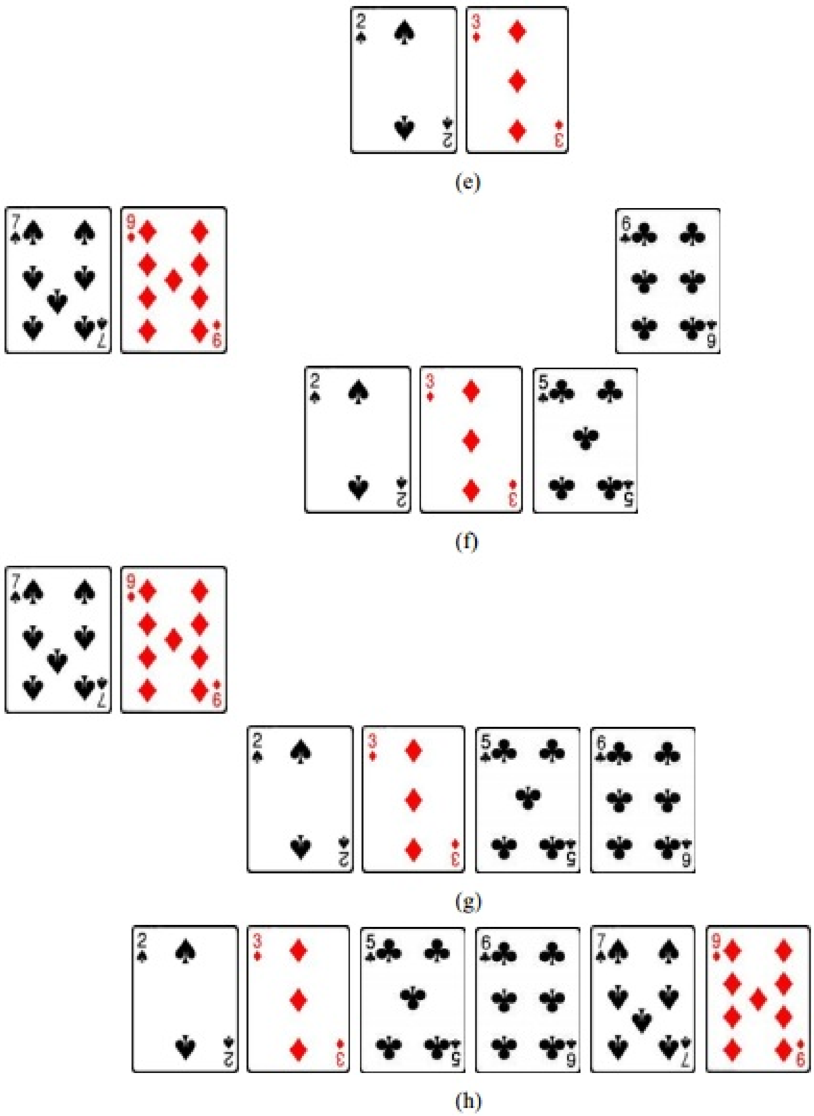


图 10.5 归并排序

下面是对图 10.5 所示过程的简要解释：

(a) 无序的初始扑克牌集合，牌太多导致难以一手进行排序；

(b) 一分为二，玩家和帮忙者两人各持一半牌；

(c) 两人分别对手中牌进行排序，从而得到两手局部有序的扑克牌序列；

(d) 两人比较各自手中的局部最小牌（黑桃 2 和方块 3），其中更小的黑桃 2 是全局最小 牌，将它放到存放归并结果的某个地方（比如桌子上）；

(e)(f)(g) 重复(d)的做法，相继将方块 3、梅花 5 和梅花 6 放到归并结果序列中；

(h) 由于第二个序列已经没有牌了，故将第一个序列剩余的牌接在归并结果序列之后。 至此形成了全局有序序列。

通过图 10.5 的形象化演示，相信读者已经理解归并过程。现在还有一个问题：图 10.5(c) 是对图 10.5(b)的两手牌分别进行“排序”后得到的，问题是怎么排序？显然，我们又回到 了初始的“排序”问题，只不过这次的排序问题具有较小的规模：初始问题是对 6 张牌排序，

现在只需两人分别对自己的 3 张牌排序。这让我们想起了“递归”这个设计利器。是的，如果觉得 3 张牌还是太多，那么可以重复上述一分为二、局部排序、全局归并的过程。这个过程可以一直进行到只有 1 张牌的情形，这时根本无需排序，因为 1 张牌自然是局部有序的。 这样就得到了递归的奠基情形，此时无需递归，只需归并。由于满足了每次递归数据规模减 小和有奠基情形这两个条件，上述递归过程是正确的。归并排序算法的伪代码如下，其中划 线部分表现了该算法的递归结构。

算法：对 datalist 执行归并排序 输入：无序的列表 datalist 输出：有序的列表 datalist 将 datalist 一分为二：list1 和 list2 对 list1 执行归并排序 对 list2 执行归并排序 归并 list1 和 list2，结果放入 datalist

下面我们用 Python 编制一个完整的程序来实现并排序算法。程序 10.1 主要由两个函数 构成：函数 merge 用于归并两个局部有序的列表 list1 和 list2，结果放在 mergelist 中；函数 mergeSort 则利用分治法和递归实现对列表 datalist 的排序。

【程序 10.1】mergesort.py

```
def merge(list1,list2,mergelist):
    i,j,k = 0,0,0
    n1,n2 = len(list1),len(list2)
    while i < n1 and j < n2:
        if list1[i]&lt;list2[j]:
            mergelist[k] = list1[i]
            i = i + 1
        else:
            mergelist[k] = list2[j]
            j = j + 1
        k = k + 1
    while i < n1:
        mergelist[k] = list1[i]
        i = i + 1
        k = k + 1
    while j < n2:
        mergelist[k] = list2[j]
        j = j + 1
        k = k + 1
def mergeSort(datalist):
    n = len(datalist)
    if n > 1:
        m = n / 2
        list1,list2 = datalist[:m],datalist[m:]
        mergeSort(list1)
        mergeSort(list2)
        merge(list1,list2,datalist)
    data = [9,2,7,6,5,3]
    mergeSort(data)
    print data
```

执行程序 10.1，将在屏幕上看到输出：

```
[2, 3, 5, 6, 7, 9]
```

顺便提醒读者注意：程序 10.1 中，函数 mergeSort 的形参 datalist 是列表类型，调用时 我们传递列表 data 作为实参。由于函数对列表类型的实参的修改后果是可以带出函数的①， 所以当我们将无序的 data 传给 mergeSort，等 mergeSort 执行完毕，data 就变成有序的了。

前面介绍的二分搜索算法其实也是分治法的应用，只不过将数据平分为两部分之后，只需"治"其中一部分，另一部分可以忽略。后面的 Hanoi 塔问题也是分治法的应用。

最后小结一下分治法。解决一个问题时，经常将问题分解为较小的问题，小问题和大问 题是同类问题。解决了小问题之后，将部分解合并，形成初始问题的最终解。如果小问题完 全类似于初始问题，只是规模较小，显然可以用递归法设计算法。

10.4 贪心法

考虑一个应用问题：假设需要在油库 A 和加油站 B、C、D、E、F、G、H 之间修建输 油管道，油库和各加油站的位置如图 10.6 所示，图中的虚线表示可能的管道铺设路线，虚 线旁标注的数值表示所需铺设的管道的长度（千米）②。例如油库 A 与加油站 B 之间需要铺 设 35 千米的管道。

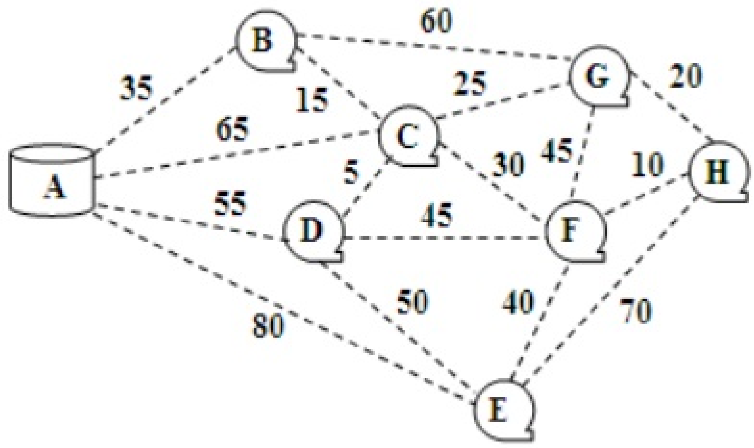


图 10.6 油库及加油站位置示意图

显然没有必要在所有可能路线上铺设管道，而只需要各加油站直接或间接与油库连通即可。假设人手和资金比较紧张，工程只能分批分期进行，每期建设一条管道。我们该如何规 划整个工程呢？

> ① 术语称为引用传递，以区别于普通的值传递。参见第 6 章。
>
> ② 此处的长度数据不一定是两点之间的直线距离，所以不要根据三角不等式（三角形中两边之和大于第三 边）得出数据不合理的结论。

指导思想当然是又快又省钱。一种想法是尽可能快地使加油站投入使用，每一期工程都 使一个加油站能够供油。那么，第一期必须在油库 A 与某个加油站之间铺设管道，问题是 选哪个加油站呢？显然应该选择 B，因为在从 A 可直接到达的 B、C、D、E 中，AB 是最短 的管道，可以在最短时间内建成，当然花费也是最少的。接下来考虑第二期工程时，可以选择一个从 A 或者 B 可到达的加油站，注意此时所选加油站不必与油库 A 直接相通，间接连 通也能保证供油。C、D、E、G 都是从 A 或 B 可通达的加油站，其中 C 是最近的，因此我 们选择 C，并铺设 B 和 C 之间的 15 千米管道。在工程的第三期，需要选择一个能与 A、B 或 C 可到达的加油站，这次最短的是 C 和 D 之间的 5 千米管道，因此选择 D 并铺设 CD 管 道。到目前为止，工程进展如图 10.7 所示，图中实线段表示已经铺设的管道，B、C、D 都 能供油了。

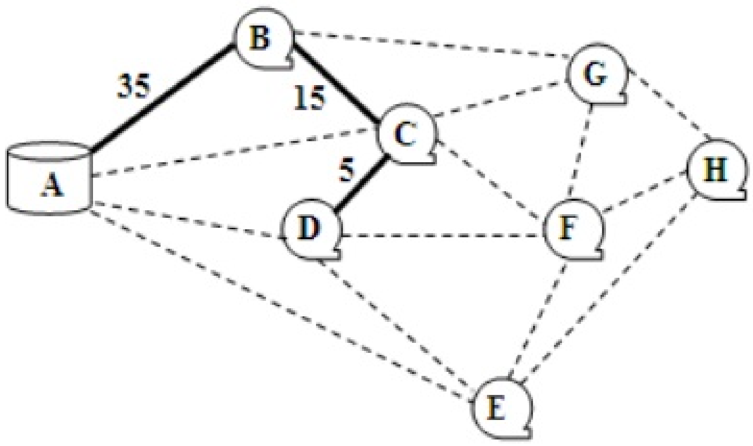


图 10.7 第三期工程后的状况 依此类推，在接下去的第四期到第七期工程中，可以分别铺设 CG、GH、FH 和 FE 之间的管道。至此，所有加油站都通过输油管道与油库 A 连通了，如图 10.8 所示。工程规划 者一定很满意，因为他们觉得自己在每一期建设中都选择了当时情况下最短的线路，从而能 以最快时间完成那一期工程，使一个新加油站投入运营。当工程完工时，铺设管道的总长度 是 150 千米。

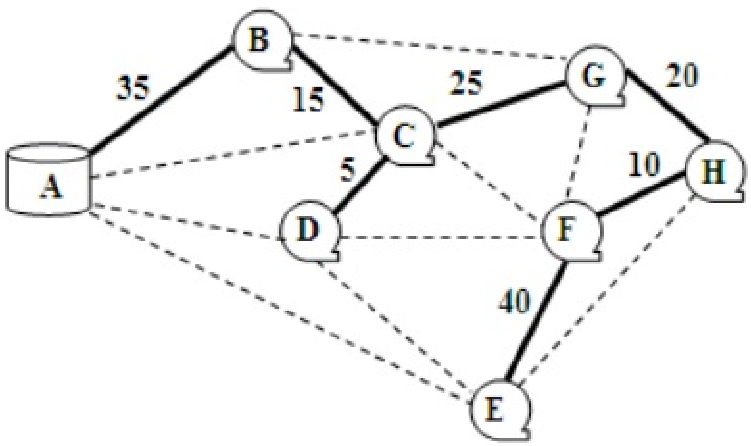


图 10.8 完工后的状况

下面考虑另一种工程建设方案。工程规划者并不追求各加油站尽快投入使用，而一心只想以最小的投资完成工程。这时的指导思想是，每一期工程都尽可能选择当前所有线路中最 短的线路来铺设管道，并确保最终能将油库和所有加油站连通起来。

按照这个思路，首先应该选择铺设 CD 管道，因为这条管道的长度是 5 千米，是所有管 道线路中最短的。完成 CD 管道之后，剩余线路中最短的管道是 10 千米的 FH，因此选择它 作为第二条铺设的管道。依此类推，接下去应该分别铺设 BC（15 千米）、GH（20 千米）和 CG（25 千米）等管道，至此工程现状如图 10.9 所示。

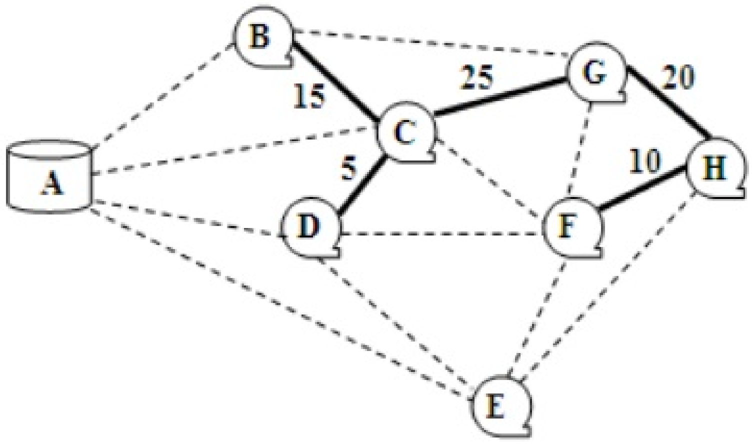


图 10.9 铺设五条最短管道之后的状况

按照上述思路接下来应该铺设当前最短的 CF 管道（30 千米），但由于 C 和 F 已经连入 了输油管道系统，再铺设 CF 管道属于重复建设，因此我们放弃 CF 而选择铺设 AB 管道（35 千米）。最后一步铺设 EF 管道（40 千米），至此油库和所有加油站都连通了，如图 10.10 所 示。

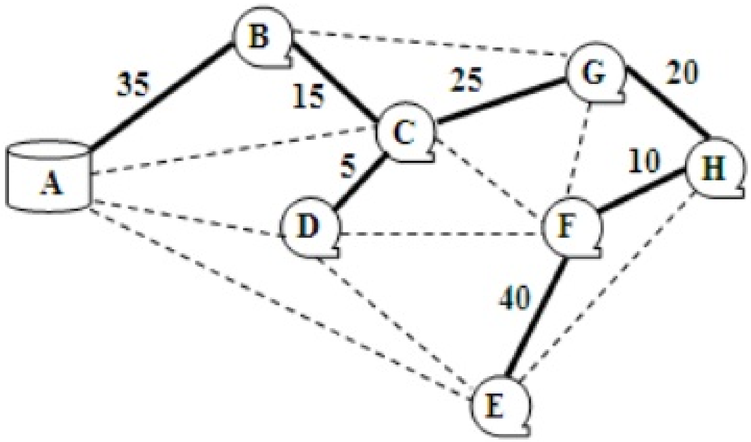


图 10.10 完工后的状况 读者一定已经发现，第二种以省钱为指导思想的建设方案与第一种以尽快投入运营为指

导思想的建设方案所导致的输油管道系统是一样的，两者都铺设了总长度为 150 千米的管道。问题是这两种建设方案到底是不是最优的呢？会不会有一种管道总长度更小的方案呢？读者不妨试试其他选择，最终会发现任何其他将油库和加油站连接在一起的方案都导致总长度超过 150 千米的管道系统。所以，我们讨论的两种方案都导致了最优的（即总长度最小）输油管道系统。

不难看出，实际中的许多问题都可以利用上述方案来解决，如下水道系统、芯片设计、 交通网、通信网等等。这些问题可以抽象成图论中的“最小支撑树”问题，上面两种解决方 案其实是解决最小支撑树问题的两个著名算法的应用。

第一种方案称为 Prim 算法，其思想是从一个地点（如油库）出发，一个接一个地将其 他地点（如加油站）连入系统，其中每一步都尽可能选择最短连接路线。Prim 算法的伪代 码如下：

```
Prim 算法
1\. 初始时所有地点标记为不可通达。
2\. 选择一个特定地点，标记为可通达。
3\. 重复下列步骤，直至所有地点都被标记为可通达：
     选择距离最近的两个地点，其中一个地点的标记是可通达，另一个地点的标记是不可通 达。然后将这两个地点连接
```

第二种策略称为 Kruskal 算法，其思想是每一步将当前距离最近且尚未连通的两个地点 连接起来。如果某一步的当前最小长度线路所涉及的两个地点已经连通了，则放弃这个路线， 接着考虑其后线路。算法伪代码如下：

```
Kruskal 算法
重复以下步骤，直至所有地点都直接或间接地连通：
     将当前距离最近并且尚未连通的两个地点连接起来。
```

Prim 算法和 Kruskal 算法虽然是不同的解决方法，但他们都能产生最小支撑树。这两个 算法其实反映了一个共同的算法设计方法——贪心法。贪心法指的是这样一种问题求解策 略：在求解过程的每一步都尽量作出在当前情况下局部最优的选择，以期最终能得到全局最 优解。例如 Prim 算法在每一步都选择当前与已连通部分最近的地点，Kruskal 算法在每一步 都尽可能选择当前最短的线路，两者的最终目标都是构造最小支撑树。

贪心算法的一般模式是通过迭代（循环）来一步一步地进行贪心选择，从而产生一个局 部最优解，并将问题简化为更小的问题，最终的全局解由所有局部解组成。即：

```
贪心算法模式
算法：
输入：一个候选对象集合
输出：由某些候选对象组成的全局解
重复以下步骤，直至得到全局解：
     从候选对象中选择当前最优者，并加入到局部解中
```

在迭代的每一步，贪心选择可以依赖于此前的迭代步骤中已经作出的选择，但不能依赖 于未来的选择。打个比方，贪心选择就像一个每次只计算一步棋的棋手，他总是选择当前能 获得最大利益的一步棋，而不考虑这步棋会不会在以后造成损失。显然，一步棋的好坏不能 只取决于当前利益，而是要着眼全局。在贪心策略下，以后即使认识到前面某一步棋不佳， 也是不允许悔棋的。可见，贪心算法具有“只看眼前利益”和“落子无悔”的两大特点。

当然，好的棋手是不会采用贪心策略来下棋的，他们会计算未来的很多步棋，然后选择 全局最优的着法。这说明贪心策略只能对某些问题（如上述最小支撑树问题）能产生全局最 优解，对另一些问题则不然。不过，贪心算法的优点是能够较快地找出解法，产生的结果经 常也是接近全局最优解的；而一心追求全局最优解则有可能导致无法在合理的时间内达到目标，就像棋手如果指望算无遗策，那就要花费大量时间来计算着法，这几乎是不可能的。

最后顺便提一下，在前面的输油管道问题中，为了从油库 A 向加油站 E 供油，采用贪 心算法设计出的方案是将 A 经 B、C、G、H、F 来与 E 连通，这条管线的总长度为 145 千 米。而假如直接在 A 和 E 之间修一条管道的话只需要 80 千米！可见，如果待解决的问题是 修建从

油库到每一个加油站的最短管道，前述两个算法是不合适的。事实上，存在另一个采 用贪心法设计的著名算法——Dijkstra 最短路径算法，可以很好地解决这个问题。

10.5 算法分析

通过前面各小节的介绍，我们看到可以设计出多种不同的算法来解决同一个问题，如搜 索问题中的线性搜索和二分搜索，排序问题中的选择排序和归并排序，最小生成树的 Prim 算法和 Kruskal 算法，等等。本节要讨论的是：解决同一问题的不同算法有好坏之分吗？

10.5.1 算法复杂度

为了回答上述问题，首先要明确如何衡量算法的好坏。以搜索问题为例，线性搜索算法

直接了当，易设计易实现，这算不算“好”？而二分搜索算法虽然设计实现稍难一些，但因 无需检查每一个数据而大大提高了搜索效率，这又算不算“好”？

在解决数学问题时，不论是证明定理还是计算表达式，只要证明过程正确、计算结果精 确，问题就可以认为成功地解决了，即正确性、精确性是评价数学解法好坏的标准。而在用 计算机解决问题时，仅仅要求算法能正确、精确地解决问题，是不够的。试想，假如一个算 法虽然能够正确地解决问题，但需要在计算机上运行 100 年或者需要占用 100TB 的内存， 这样的算法有实际意义吗？不要以为 100 年或 100TB 是危言耸听，很多简单算法都可能轻 易地达到或突破这样的需求（参见稍后对 Hanoi 塔算法的分析）。可见，利用计算机解决问 题必须考虑算法的经济性，即算法所耗费的资源问题。计算机用户当然都希望能多快好省地 解决问题，因此好的算法应当尽量少地耗费资源。

通常只考虑算法所占用的 CPU 时间和存储器空间这两种资源①。所谓算法分析，就是分 析特定算法在运行时所耗费的时间和存储空间的数量，分别称为算法的时间复杂度和空间复 杂度。本节只讨论算法的时间复杂度，毕竟存储空间的大小在现代计算机中已经越来越不再 是一个问题。

虽然讨论的是算法耗费的“时间”，但我们并不是真的去测量程序在计算机中的实际运 行时间，因为实际运行时间依赖于特定机器平台（CPU、内存、操作系统、编程语言等）， 同一算法在不同平台上执行可能得到不同的分析结果，故很难据此对算法进行分析和比较。 例如，在最先进的计算机上执行线性搜索也许比在老式计算机上执行二分搜索要快，据此得 出线性搜索优于二分搜索显然不合理。

实际上，算法分析指的是分析算法的代码，估计出为解决问题需要执行的操作（或语句、 指令等类似概念）的数目，或称算法的“步数”。之所以分析算法步数，是因为：第一，步 数确实能反映执行时间——步数越多执行时间就越长；第二，算法的步数不依赖于平台，更 容易分析和比较。

例如，下面的函数 f1()需要执行 11 次赋值操作，其中包含 10 次加法运算②：

```
def f1():
    x = 0
    for i in range(10):
        x = x + 1
```

而下面的函数 f2()需要 21 次赋值操作（20 次加法）：

```
def f2():
    x = 0
    for i in range(20):
        x = x + 1
```

比较一下 f1 和 f2，显然 f1 运行时间更短，但这并不意味着 f1 比 f2 采用的算法“好”， 因为它们的“算法”显然是一样的，只不过 f1 要处理的数据更少：f1 将 10 个 1 相加，而 f2 将 20 个 1 相加。可见，算法复杂度是跟算法处理的数据量有关的。

算法通常都设计成能处理任意大小的输入数据，这就导致算法的步数并不是固定的，而 是随着问题规模的变化而变化，因此算法的步数可表示为问题规模的函数。假设用 n 表示问题规模，算法分析不仅要考虑算法步数与 n 的关系，更重要的是还要考虑“当 n 逐渐增大时” 算法复杂度会如何变化。例如，将上述 f1 和 f2 改写成更一般的形式：

> ① 不考虑开发算法的人力物力等代价。
>
> ② 注意我们分析的层次是源代码级别，而不是机器指令级别。

```
def f(n):
    x = 0
    for i in range(n):
        x = x + 1
```

不难得出此函数需要执行的步数为 n+1。当 n 增大时，算法执行时间也会增加，而且是线性地增加，即：当 n 增加 1 倍变成 2n，执行时间变成 2n+1，大约比 n+1 增加 1 倍。

说 A 算法比 B 算法好，并不是指对于特定的 n，A 比 B 节省 50%的时间，而是指随着 n 的不断增大，A 对 B 的优势会越来越大。

算法复杂度的大 O 表示法

再次观察上面例子中函数 f()的步数表达式“n+1”，不难看出其中对执行时间起决定作 用的是 n，而 n 后面的+1 是可以忽略不计的。按照“当 n 逐渐增大时”进行分析的思想， 即便是 n+100、n+1000000 中，n 后面的常数也是可以忽略不计的，因为与逐渐增大趋于∞ 的 n 相比，任何常数都是浮云。事实上，分析算法复杂度时，我们只分析其增长的数量级， 而不是分析其精确的步数公式。

数学中的“大 O 表示法”根据函数的增长率特性来刻画函数，可以用来描述算法的复 杂度。令 f(n)和 g(n)是两个函数，如果存在正常数 c，使得只要 n 足够大（例如超过某个 n0）， 函数 f(n)的值都不会超过 c×g(n)，即当 n > n0 时，

```
f(n) <= c x g(n)
```

则可记为

```
f (n) = O(g(n))
```

在描述算法复杂度时，n 对应于问题规模，f(n)是算法需执行的步数，g(n)是表示增长数 量级的某个函数。说算法的复杂度为 O(g(n))，意思就是当 n 足够大时，该算法的执行步数（时间）永远不会超过 c×g(n)。

例如，假设一个算法当输入规模为 n 时需要执行 n+100 条指令，则当 n 足够大时（只 要大于 100），

```
n + 100 <= n + n = 2n
```

套用大 O 表示法的定义，取 g(n)=n，则可将此算法的复杂度表示为 O(n)。同理，如果一个算法的步数为 n+1000000，它的复杂度仍然可表示为 O(n)。由此可见，两个不同的算法虽然具有不同的代码和执行步数，但完全可能具有相同的复杂度，即当问题规模足够大时，它们的执行时间按相同数量级的增长率增长，利用大 O 表示法即可描述这一点。

实际分析算法时，为了使 O(g(n))中的 g(n)函数尽量简单，在得到算法的步数表达式 f(n) 之后，可以利用两条规则来简化推导，直接得出 f(n)的大 O 表示。规则如下：

（1）如果 f(n)是若干项之和，则只需保留最高次项，省略所有低次项；

（2）如果 f(n)是若干项之积，则可省略任何常数因子（即与 n 无关的因子）。 例如，分析下列代码：

```
def f(n):
    x = 0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                x = x + 1
    for i in range(n):
        for j in range(n):
            for k in range(n):
                x = x + 1
    for i in range(n):
        x = x + 1
```

易知此算法的步数为 $2n^3$+n+1。根据第一条规则，可只保留 2n3；再根据第二条规则，可只保留 n^3。所以，此算法的复杂度为 O(n^3)。当然我们也可以直接从 f(n)开始推导，利用大 O 表示法的定义来验证这个结果是正确的：对于 n > 1，

$$f(n) = 2n^3 + n + 1 \le 2n^3 + n^3 + n^3 = 4n^3$$

取 g(n)为 n3，c 为 4，即得 f(n) = O(n3)。 总之，以上两条规则告诉我们，在分析算法代码时可以忽略许多代码，而只关注那些嵌套层数最多、并且每一层循环的循环次数都与问题规模 n 有关的循环。

10.5.2 算法分析实例

本节以本章介绍的若干算法为例来讨论对算法复杂性的分析。

搜索问题的两个算法 对于搜索问题，本章介绍了线性搜索和二分搜索两个算法。

线性搜索算法的思想是逐个检查列表成员，编码时可以用一个循环语句来实现。循环体 的执行次数取决于列表长度：如果列表长度为 n，则循环体最多执行 n 次。因此，如果列表 长度增大一倍，则循环次数最多增加一倍，算法执行的步数或实际运行时间最多增加一倍。 可见，线性搜索算法在最坏情形下的运行时间与输入列表的大小 n 呈线性关系，即复杂度为 O(n)，称为线性时间算法。

二分搜索算法的主体也是一个循环，但该循环不是逐个检查列表数据，而是每次检查位 于列表中点的数据，并根据该中点数据与要查找的数据的大小比较情况来排除掉左半列表或 右半列表。接着对保留下来的一半列表重复进行这个"折半"过程。显然，循环的次数取决 于输入列表能"折半"多少次。如果初始输入列表有 16 个数据，则第一轮循环后剩下 8 个数据，第二轮循环后剩下 4 个数据，第三轮后剩下 2 个，第四轮后只剩下 1 个数据。因此， 最多四轮循环后就能得出搜索结论：要么找到，要么不存在。一般地，如果输入规模为 n， 则二分搜索算法最多循环 log2n 次，即复杂度为 O(log2n)，称为对数时间算法。要说明的是， O(log2n) 表示复杂度与问题规模 n 的对数成正比，至于这个对数是以 2 为底还是以 10 为底并 不重要，因此我们经常省略对数的底，写成 O(log n)。

O(n)与 O(log n)到底有多大差别？回到 10.2 中提到的猜数游戏，假如某甲心中想好一个1百万以内的数让某乙来猜。某乙从小到大逐个试猜（即线性搜索）的话，运气好猜 1 次就能命中，运气不好最多要猜 1 百万次。平均来说需要猜 50 万次才能猜中。而如果某乙每次 猜中间数（即二分搜索）的话，则最少猜 1 次，最多也不过猜 log21000000≈20 次就能猜中。 可见，随着 n 的增大，O(log n)远远优于 O(n)。

排序问题的两个算法

对于排序问题，本章介绍了选择排序和归并排序两个算法。

首先推导选择排序算法的步数与问题规模（即数据列表的长度）的关系。选择排序算法 首先找出全体数据中的最小值，并将该值作为结果列表的第一个成员。其次，算法从剩余数 据中找出最小值，并将该值作为结果列表的第二个成员。依此类推，直至产生有序列表。假 设列表初始大小为 n，为找出最小值，算法需检查每一个数据。接下来算法从剩余 n-1 个数 据中找出最小值，这需要检查 n-1 个数据；第三次循环从 n-2 个剩余数据中找出最小值。这 个过程一直继续到只剩 1 个数据为止。因此，选择排序需要执行的步数为

$$n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 = \frac{1}{2}n^2 + \frac{1}{2}n$$

按照前述规则，可以看出选择排序算法所需的步数与数据列表大小的平方成正比，即算法复杂度为 O(n^2)，称为二次方时间算法。 其次，我们来推导归并排序算法的步数与列表大小的关系。归并排序算法的基本思想是将列表一分为二，然后对两半数据各自排序，最后再合并成一个列表。其中对两个子列表的 排序又是通过递归调用归并排序来实现的，最终将分解到长度为 1 的列表，这时可直接进行 归并。由此可见真正的排序工作是在归并过程中完成的，该过程所做的只是将来自子列表的 数据按从小到大的顺序逐个复制到初始列表的合适位置。图 10.11 展示了对列表[0,5,7,2]进 行归并排序的过程。图中用虚线表示初始列表的递归分解过程，逐步分解后最终得到长度为 1 的列表。这些长度为 1 的列表再进行归并，逐步形成长度为 2、4 的有序的列表，图中用实线箭头表示归并时各数据的逐步到位过程。从图 10.11 容易分析出归并排序算法的步数。 从左向右，分解过程并不比较数据大小来排序，这部分工作可以忽略。接下来的归并过程包 含大量比较、复制操作，是整个算法的工作量的体现。归并过程分为 log2n 层，以逐步形成 长度为 2、22、23、…、n 的有序子列表①。又因为每一层归并都需要对全部 n 个数据进行处 理，所以归并排序算法的步数是“n×层数”，即具有复杂度 O(nlog n)，可称为 nlog n 时间 算法。

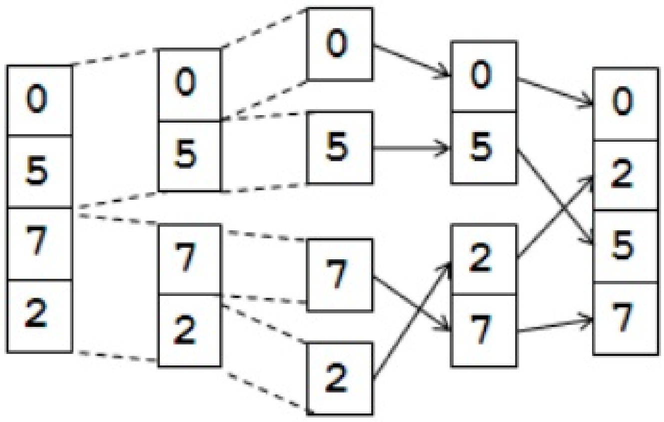


图 10.11 归并排序过程示意图

n^2 与 nlog n 有多大差别呢？当 n 较小时，两者差距不大，选择排序算法甚至有可能还快 一些，因为它的代码更简单。但是，随着 n 的增大，log n 只是缓慢地增大，因此 n×log n 的增长速度远远低于 n×n。这就是说，对于大量数据，归并排序算法的性能远远好于选择 排序算法。

> ① 如果 n 不是 2 的幂，子列表的长度当然也不会都是 2 的幂。

Hanoi 塔算法

下面推导 Hanoi 塔问题的递归算法的步数与圆盘个数 n 的关系。与基于循环（迭代）的算法不同，递归算法不容易直接从代码形式上看出具体的操作步数。对于 Hanoi 塔递归算法， 我们可以直接考虑将 n 个圆盘从 A 柱移到 C 柱所需的移动次数。

根据算法的结构，为了移动 n 个圆盘，需要先将 n-1 个圆盘从最大圆盘上移开，然后移 动最大圆盘，最后再将 n-1 个圆盘移到最大圆盘上。假设 f(n)是移动 n 个圆盘所需的步数， 则应用一点中学数学知识很容易推导出

$$
\begin{aligned}
f(n) &= f(n-1)+1+f(n-1) \\
&= 2f(n-1)+1 \\
&= 2^2 f(n-2)+2+1 \\
&= 2^3 f(n-3)+2^2+2+1 \\
&= \ldots \\
&= 2^{n-1}+2^{n-2}+\ldots+2^2+2+1 \\
&= 2^n-1
\end{aligned}
$$

可见，Hanoi 塔算法的复杂度为 O(2n)，称为指数时间算法，这是因为问题规模的度量 n 出现在步数公式的指数部分。

指数时间算法到底有多复杂呢？读者也许听说过“指数爆炸”这个名词，它表明指数时 间算法所需要的执行时间会随着问题规模的增长而迅速增长。在 Hanoi 塔故事中，即使僧侣 们 1 秒钟就能移动一步圆盘，并且每天都不休息，为了移动 64 个圆盘，也需要花费 264－1秒，即 5850 亿年！可见，指数时间算法只适用于解决小规模的问题。

总之，利用计算机解决问题时，需要考虑算法的时间复杂性，这是衡量问题难度和算法 优劣的一个重要指标。有些应用对于运行时间有较高要求，运行时间过长的话可能导致计算 结果过时、失效。图 10.12 给出了本章见过的各种算法复杂度的大致比较，图中横坐标表示 问题规模 n，纵坐标是算法执行时间（或步数）。虽然图中曲线不是很精确，但足以说明指 数时间和二次方时间算法是多么不适合大量数据，而其他几种复杂度的曲线则相当平缓。

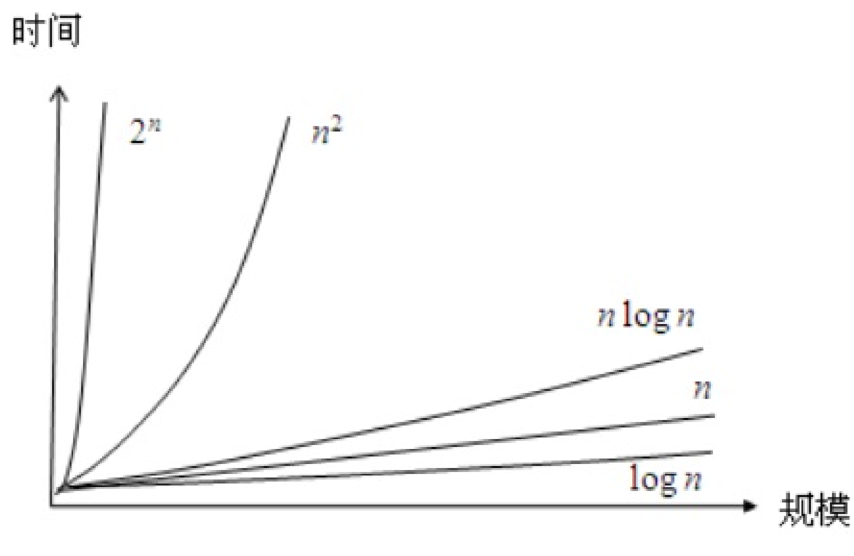


图 10.12 各种算法复杂度比较

10.6 不可计算的问题

到目前为止，我们讨论的所有问题都是可解的。有些问题的解法非常有效，有些问题的

解法则比较复杂。Hanoi 塔之类的问题称为难解问题，因为当问题规模较大时，相应算法需要太多太多的时间（或空间）来完成计算，事实上是无效、不可行的解法。

现实中还存在比难解问题更麻烦的问题，那就是不可解问题。考虑这个场景：计算机正在执行一个程序，我们坐在边上等待程序结束。当过了很久程序还没结束时，我们该怎么办 呢？我们可能推测程序中出现了无穷循环，永远不会结束，这样我们就必须强行中断程序运 行甚至重启计算机。然而，我们并不能绝对肯定是出现了无穷循环，也许是因为计算太复杂 导致时间过长呢？这样的话，我们就该继续等待。显然，这是一个两难困境。我们设想，要 是有这么一个程序 P 就好了：P 的功能是以另一个程序 Q 的代码作为输入，并分析 Q 中是 否包含无穷循环。然而很遗憾，这样的程序 P 是不存在的！这个问题其实对应于图灵机的 停机问题，下面对此进行简要介绍。

图灵机

英国数学家 Alan Turing 于 1936 年发明了一种抽象机器用于研究计算的本质，人们称这 种机器为图灵机（Turing machine）。图灵机能够模拟算法式计算，即按预定的规则一步一步 执行基本指令的过程。现代计算机就是这样按照预定的程序一步一步执行指令的，因此可以 视为图灵机的具体实现。

人们为了进行计算，需要用到纸和笔。类似地，图灵机在“硬件”上由一条纸带和一个 读写头组成：纸带用于记录信息，读写头用于读写信息。纸带在读写头下移动，读写头即可 在纸带上写下符号（如 0 和 1）或读出符号。这有点类似磁带录音机中磁带与磁头的关系， 但与录音机的顺序录音或回放不同的是，图灵机的读写头和纸带受预定的规则（相当于我们 熟悉的程序）的控制。参见图 10.13。



图 10.13 图灵机的纸带和读写头

下面对图灵机进行更详细的描述。一个图灵机涉及以下一些要素：

- 纸带：纸带被划分成一个个格子单元，单元中可以写入符号。纸带在向左、向右两 个方向上都是无限延伸的，即图灵机的存储能力不受限制。

- 读写头：用于读写纸带单元中的符号。纸带在读写头下可以向左或向右移动，每次 移动一个单元。当然也可以理解成纸带不动而读写头左右移动。
- 符号表：能够写入纸带的合法符号。具体用什么符号系统并不重要，正如现代计算 机基于二进制一样，只要提供 0 和 1 两个符号就足够从事任何计算。
- 状态：图灵机在任一时刻都处于某种状态，例如当前读写头下方是 0 或 1 即对应不 同状态。不同状态的数目是有限的。两个特殊状态分别是开始状态和停止状态。
- 指令：指令描述的是如何根据图灵机的当前状态以及当前读写头所读到的符号来控 制图灵机执行特定动作并转换为新的状态。形如：

当前状态，输入符号 ® 新状态，输出符号，移动读写头 预定的多条指令构成一个指令表（程序），它完全决定了图灵机的行为。图灵机的 运行就是按照指令表所确定的状态转换规则一步一步进行状态转换的过程。

下面我们设计一个对给定正整数 n 加 1 的图灵机。

【图灵机 T+1】T+1 的符号表仅由 0（表示空白）和 1 组成。正整数 n 在纸带上用 n 个连续单 元的 1 表示，例如 1、2、3 在纸带上分别表示为 1、11、111。读写头初始位置是在输入数 据 n 的左方，停止位置是在输出数据 n+1 的最后一位 1 之上。初始状态为 s1，停止状态为 s3。指令表如下：

```
s1, 0 => s1, 0, R
s1, 1 => s2, 1, R
s2, 0 => s3, 1, Stop
s2, 1 => s2, 1, R
```

假设输入数据是 3，则图 10.14 展示了 3+1 的计算过程。读写头里记录的是图灵机当前 状态。

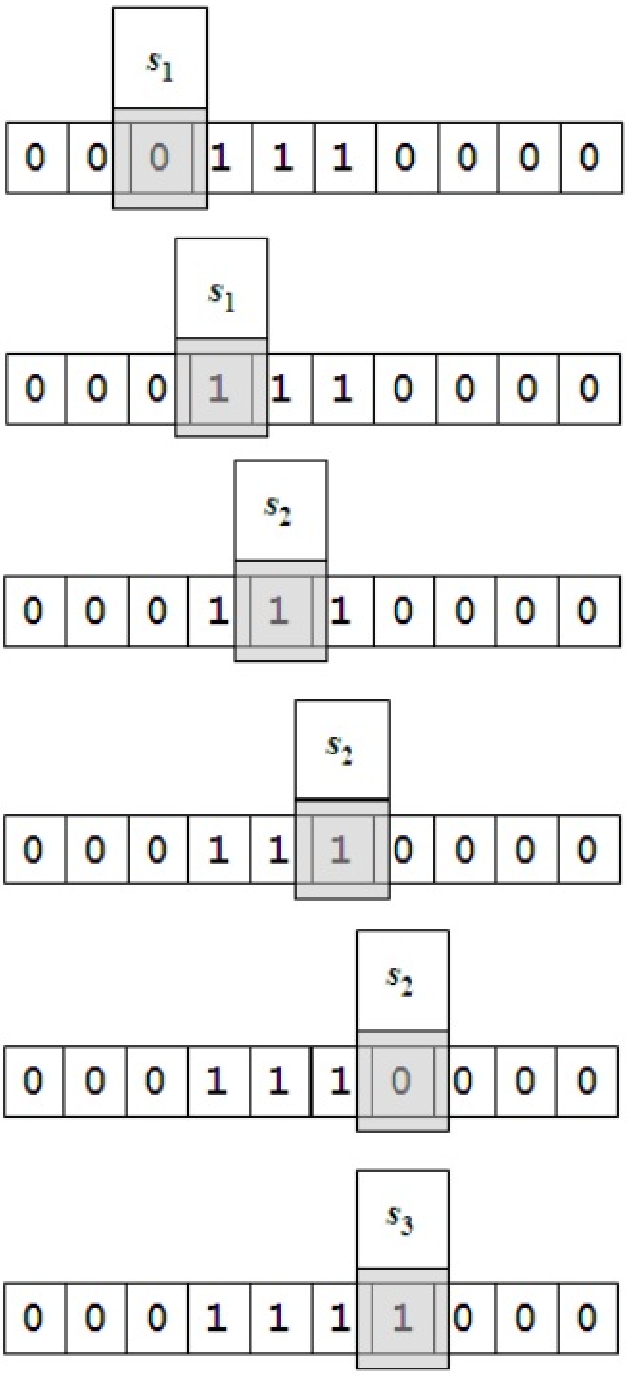


图 10.14 计算 n+1 的图灵机 T+1（输入 n=3）

第 1 条指令的意义是：当图灵机 T+1 处于 s1 状态，并且读写头所读单元的内容是 0，那 么就保持 s1 状态，也不改动该单元的内容，然后读写头右移。第 3 条指令的意义是：当 T+1 处于 s2 状态，并且读写头所读单元里的内容是 0，那么就进入 s3 状态，将该单元内容改为 1， 然后停止。其他两条指令的意义请读者自行解读。从初始状态开始执行这些指令，经过 6 步状态转换，T+1 将终止，并且终止时纸带上的计算结果是 4 个连续的 1，表示正整数 4。

尽管图灵机是如此简单，但它的计算能力却非常强大。从上例可知，存在计算 n+1 的图灵机，由此不难想象可以设计出计算 n+m 的图灵机，进而可以设计出计算 n×m 的图灵 机，等等。注意，这里我们谈论图灵机的计算能力，并非针对它的计算速度或存储空间，因 为图灵

机毕竟不是现实的计算机。研究图灵机是为了在理论上探索计算的能力和局限，例如 回答计算机科学的一个根本问题：究竟什么是可计算的？对此，Turing 和 Church 分别通过 研究图灵机和λ演算，得出了一个重要假设——Turing-Church 论题，其大致意思是：一个 问题是能行可计算的（即算法可计算），当且仅当该问题能用图灵机来计算。因此，图灵机 事实上给出了“算法计算”或“机械计算”的精确意义。

图灵机的强大计算能力有一个重要表现，那就是一个图灵机可以模拟另一个图灵机的工 作。如果将图灵机 T 1 的功能进行编码，然后输入给另一图灵机 T 2，那么 T 2 就能表现得像 T 1 一样。打个比方，这就像一个人可以模拟另一个人的行为一样。假设张三既懂加法又懂 乘法，并且他知道不懂乘法的李四总是错误地将 n×m 算成 n+m，那么当我们将 n 和 m 输 入给张三要他计算 n×m 时，他完全可以故意输出 n+m 来冒充李四。

既然一个图灵机可以模拟另一个图灵机的行为，那我们就可以设计一个通用图灵机，它 可以模拟任何图灵机的行为。对此读者应不陌生，因为我们在第 1 章就说过，现代计算机是通用计算机，给它安装不同的程序，就能完成不同的功能。图 10.15 展示了如何用通用图灵 机 UT 来模拟某个特定图灵机 T：将 T 的行为（指令表）用 0/1 序列进行编码得到 Tcode， 连同 T 的输入数据 data 一同输入给 UT，然后 UT 即可对 Tcode 进行解码，并针对 data 来模 拟 T 的行为。

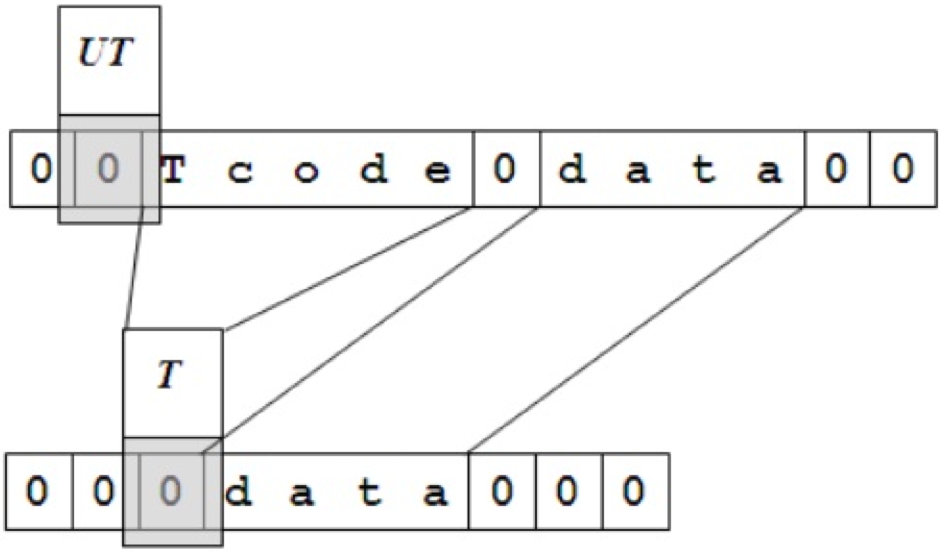


图 10.15 通用图灵机 UT 模拟特定图灵机 T

如果用函数来表示图灵机，则 UT 模拟 T 的行为可表示为

```
UT(Tcode,data) = T(data)
```

停机问题

对任何给定的图灵机 T，以及输入数据 data，T 可能停机，也可能不停机。上面计算 n+1 的图灵机 T+1 显然总是能停下来的，因为正整数 n 在纸带上表示为 n 个连续的 1，T+1 的第二条指令要求读写头只要读到 1 就不断向右移，因此最终会读完这有限个数的 1，并读到连续 1

右方的第一个 0，第三条指令会将这个 0 改写为 1，并停机。

一个图灵机也很容易不停机。例如这样一条指令就有可能令图灵机无法终止：

```
s1, 0 => s1, 0, NoMove
```

即，在 s1 状态读到 0 时，保持 s1 状态和单元内容 0 不变，并且读写头也不移动。如果一个图灵机进入到 s1 状态并且恰好读到 0，那么这个图灵机就将永远处于这个状态而不停机。不难看出，这条指令的行为与 Python 中的无穷循环语句

```
while True:
    pass
```

是一样的。顺便说明一下，pass 是 Python 语言的一条语句，功能是什么都不做。

我们当然希望设计的图灵机能正确地完成计算并停机，可现实经常不能如我们所愿，就 像我们编写的程序经常陷入无穷循环而不能终止一样。更让人烦恼的是，当图灵机（或程序） 一直在执行而不终止时，我们并不知道它是否陷入无穷循环了！现实中，我们只能通过运行 时间长短的经验来判断到底是什么情况，但这毕竟是不可靠的。有没有办法来检验图灵机是 否停机呢？也就是说，能不能设计这样的通用图灵机 HT，它的输入是另一个图灵机 T（的 编码）和 T 的输入 data，它的功能是判断 T 在 data 上执行后是否停机：如果是，则 HT 输出 1 并停机；如果不是，则 HT 输出 0 并停机。亦即，HT 是判断其他任意图灵机是否终止的 图灵机。

上述 HT 是否存在？这就是所谓停机问题（Halting problem）。Turing 的一个重要成果就 是证明了 HT 不存在！下面我们用程序设计的术语来非形式地描述这个证明。

从程序设计角度看，停机问题就是要编一个程序 halt，它读入另一个程序 prog 的源代 码，并判断 prog 是否导致无穷循环。由于 prog 的行为不仅依赖于它的源代码，还依赖于它 的输入数据，因此为了分析 prog 的终止性，还要将 prog 的输入数据 data 交给 halt。由此可 得 halt 的规格说明：

```
程序：停机分析程序 halt；
输入：程序 prog 的源代码，以及 prog 的输入数据 data；
输出：如果 prog 在 data 上的执行能终止，则输出 True，否则输出 False。
```

读者也许会觉得向程序 halt 输入另一个程序 prog 作为处理对象有点不可思议，但其实 这是非常普通的事情。例如，编译器（或解释器）就是这样的程序：将一个程序 P 的源代 码输入给编译器程序 C，C 可以分析 P 中是否有语法错误，有则报错，没有则输出 P 的目标 代码。

在停机问题中，正常情况下是想运行 prog(data)，但又不知道这个执行过程能不能终止， 于是希望将 prog 的代码和 data 交给停机分析程序 halt，由 halt 来判断 prog(data)的终止性。由 halt 的程序规格可见，halt 总是能得出结论并终止的，从而避免了直接执行 prog(data)时无法确切知道它是否能终止的困扰。

设计 halt 程序的初衷可以理解，可惜这个程序是编不出来的。我们用反证法来证明这个 结论，即先假设存在程序 halt，然后推导出矛盾来。

假如我们已经设计出了停机分析程序 halt，其参数是两个字符串：prog 是被分析的程序 的源代码，data 是 prog 的输入数据。

```
def halt(prog,data):
    …… # 分析 prog 代码，如果对 prog 输入 data 时运行能终止
    return True
    …… # 如果 prog 运行在 data 上不能终止
    return False
```

利用 halt 的功能，我们可以编出如下这个奇妙的程序：

```
def strange(p): result = halt(p,p)
    if result == True: # 即 p(p)终止
        while True:
            pass
    else: # 即 p(p)不终止
        return
```

运行 strange(strange),结果如何?

函数 strange()有一个字符串类型的形参 p，调用时需传递一个程序给它，不妨假设所传 递的程序也是以一个字符串数据作为输入。strange 首先调用 halt(p,p)，这里的关键技巧是， 传递给函数 halt 的形参 prog 和 data 的实参都是 p，亦即我们要分析程序 p 以它自己为输入 数据时——即 p(p)——运行是否终止。strange 根据 halt(p,p)的分析结果来决定自己接下去怎 么做：如果结果为 True，即 p(p)能终止，则 strange 进入一个无穷循环；如果结果为 False，即 p(p)不终止，则 strange 就结束。

strange 程序看上去有点费解，但只要 halt 存在，strange 在编程方面显然没有任何问题。 接下来是证明过程的最美妙的部分：假如将 strange 自身的源代码输入给 strange 时会发生什么？更确切地，strange(strange)能否终止？

我们参照上面的 strange 代码来分析。假如调用 strange(strange)不终止，那必然是因为 执行到了代码中条件语句的 if result == True 部分，即 halt(strange,strange)返回了 True，这又 意味着 strange 以 strange 为输入时运行能终止！另一方面，假如调用 strange(strange)能终止， 那必然是因为执行到了条件语句的 else 部分，即 halt(strange,strange)返回了 False，这又意味 着 strange 以 strange 为输入时运行不能终止！总之，我们得到了如下结论：

若 strange(strange)不终止，则 strange(strange)终止；若 strange(strange)终止，则 strange(strange)不终止。

这样的结论在逻辑上显然是荒谬的。导致这个矛盾的原因在于我们假设了 halt 的存在，并利用了 halt 的功能。至此，我们证明了编写 halt 程序是不可能完成的任务，即停机问题是一个不可解问题。

停机问题不可解的证明过程具有非常深刻的意义，它告诉我们算法式计算具有本质上的 局限性。计算机虽然在各行各业解决了很多问题，但是确实存在计算机不能解决的问题。

10.7 练习

1. 程序设计：找出最小自然数 n，n 满足条件“用 3 除余 2，用 5 除余 3，用 7 除余 4”。

2. 设计递归算法来解决问题：求无序数值列表 L 的最大值和最小值。

3. 改进线性搜索算法：在开始查找 x 之前，先在列表尾添加 x。这样查找 x 总能成功，但若返回的索引是列表尾，则意味着原列表中没有 x。分析、比较这个改进版本与原版本的性能。

4. 假如将“为问题 P 设计算法”本身作为问题，这个问题有没有算法？

第 11 章 计算+X

当代科学研究有三大支柱：理论、实验和计算。计算机技术的发展，为利用计算手段来 解决科学和工程问题提供了强大的支持。越来越多的领域（包括自然科学和社会科学领域） 利用计算来解决问题，将解决问题的方法从过去的定性分析发展成如今的定量计算。科学领 域与计算的结合促成了多种交叉学科的形成，如计算数学、计算物理学、计算化学、计算生 物学、计算材料学、计算经济学、计算语言学、计算考古学、计算犯罪学、计算免疫学等等。各学科下面的子学科冠以"计算"前缀的更是数不胜数，如计算数学下面的计算几何、计算 数论、计算拓扑学，计算语言学下面的计算语义学、计算词汇学、计算幽默学等。

本章介绍一些典型的"计算+X"，以使读者初步了解计算和计算思维是如何帮助各专业 领域求解问题的。

11.1 计算数学

计算数学是关于通过计算来解决数学问题的科学。这里所说的“计算”既包括数值计算， 也包括符号计算；这里所说的“数学问题”可能来自纯数学，更可能是从各个科学和工程领 域抽象出来的。计算数学包括很多分支，其中最核心、应用最广的是数值方法。

数值方法

数值方法（numerical method，也称计算方法、数值分析等）是利用计算机进行数值计 算来解决数学问题的方法，其研究内容包括数值方法的理论、分析、构造及算法等。很多科 学与工程问题都可归结为数学问题，而数值方法对很多基本数学问题建立了数值计算的解决 办法，例如线性代数方程组的求解、多项式插值、微积分和常微分方程的数值解法等等。

数值方法的构造和分析主要借助于数学推导，这是数学思维占主导的部分。例如，一元 二次方程的求根公式实际上给出了方程的数值解法，该公式完全是通过数学推导得出的；而 通过对该公式的分析，可以了解实数根是否存在等情形。如果问题不存在有限的求解代数式， 可以通过数学推导来寻求能得到近似解的代数式，例如将积分转化为求和。

数值方法最终要在计算机上实现，这是计算思维占主导的部分。有人也许会认为，对于 数值计算问题，只要有了求解问题的数学公式，再将这些公式翻译成计算机程序，问题就迎 刃而解，所以数值方法的关键是数学推导，而计算思维在其中并没有什么作用。是不是这样 呢？仍以一元二次方程 ax2+bx+c=0 的求解问题为例。这个问题的求解求根公式是已知的：

$$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

这个公式可以直接了当地翻译成 Python 程序（程序 3.5）：

```
import math
a, b, c = input("Enter the coefficients (a, b, c): ")
discRoot = math.sqrt(b * b - 4 * a * c)
root1 = (-b + discRoot) / (2 * a)
root2 = (-b - discRoot) / (2 * a)
print "The solutions are:", root1, root2
```

下面是此程序的一次执行结果：

```
Enter the coefficients (a, b, c): 1,-(9+10**18),9*10**18
The solutions are: 1e+18 0.0
```

可见，计算机求解方程 x**2 -(9+10**18)x + 9x10**18 = 0 所给出的根是 10**18 和 0，而非正确的 10**18 和 9。对于这个结果，传统的数学是无法解释的，只有明白了计算机的能力和限制，才能给出解释。计算思维在计算方法中的意义，由此可见一斑。 利用数值方法解决科学与工程问题大体要经过三个步骤。第一步是为问题建立数学模型，即用合适的数学工具（如方程、函数、微积分式等）来表示问题；第二步是为所建立的 数学模型选择合适的数值计算

方法；第三步是设计算法并编程实现，这里要着重考虑计算精 度和计算量等因素，以使计算机能够高效、准确地求解问题。在计算机上执行程序得到计算 结果后，若结果不理想，多半是因为所选数值方法不合适，当然也可能是数学模型不合适。 在模型正确、编程正确的前提下，计算结果完全取决于数值方法的选择。

本节只简单介绍计算机的能力和限制是如何影响计算方法的选择的。

误差

正如前述一元二次方程求解例子所显示的，一个正确的数学公式在计算机上却得不到正 确的、精确的结果，这种现象主要是由误差引起的。科学与工程计算中的误差有多种来源， 其中建立数学模型和原始数据观测两方面的误差与计算方法没有关系，与计算方法有关的是 截断误差和舍入误差。

截断误差是在以有限代替无限的过程中产生的，例如计算 ex 的泰勒展开式

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots$$

时只能选取前面有限的 n 项，得到的是 ex 的近似值，前 n 项之后的部分就是截断误差。 舍入误差是因计算机内部数的表示的限制而导致的误差。在计算机中能够表示的数与数

学中的数其实是不一样的：计算机只能表示有限的、离散的数，而数学中的数是无限的、连续的。以有限表示无限，以离散表示连续，难免造成误差。例如 Python 中有如下出人意料的数值计算结果：

```
>>> 1.2 - 1
0.19999999999999996
```

由于浮点数内部表示的限制，1.2 - 1 的结果并非精确的 0.2。又如，积分计算问题

是连续系统问题，由于计算机不能直接处理连续量，因此需要将连续的问题转化为离散的问题来求解。一般常用离散的求和过程来近似求解积分①。

$$\sum_{k=1}^{n} A_k f(x_k)$$

舍入误差的控制

计算机内部对数的表示构成一个离散的、有限的数集，而且这个数集对加减乘除四则运算是不封闭的，即两个数进行运算后结果会超出计算机数集的范围。这时只好用最接近的数 来表示，这就带来了舍入误差。因此，应当控制四则运算的过程，尽量减小误差的影响。

在加减法运算中，存在所谓“大数吃小数”的现象，即数量级相差较大的两个数相加减 时，较小数的有效数字会失去，导致结果中好像没做加减一样。例如

```
>>> 10**18 + 9.0
1e+18
```

> ① 据说积分号就是从 S（sum）演变而来的符号。

由此可知，当有多个浮点数相加减时，应当尽量使大小相近的数进行运算，以避免大数"吃"小数。例如，设 x1 = 0.5055×104，x2 = x3 = ... = x11 = 0.4500（假设计算机只能支持 4 位有效数字），要计算诸 xi 的总和。一种算法是将 x1 逐步与 x2 等相加，这样每次加法都是大 数加小数，按计算机浮点计算的规则：x1 + x2 ＝ 0.5055×104 + 0.000045×104 ＝ 0.505545×104＝0.5055×104，即产生了舍入误差 0.45。如此执行 10 次加法之后，结果仍然是 0.5055×104，误差积累至 10×0.45 ＝ 4.5。另一种算法是让相近数进行运算，如 x11 + x10 = 0.9000， 在一直加到 x1，执行 10 次加法之后得到总和 0.5060×104，没有舍入误差。这个例子再次显 示了"次序"在计算中的重要意义：数学上毫无差别的两种次序在计算机中却带来截然不同 的结果，就像我们在第 3 章中计算 231-1 时采用 230-1+230 这个次序一样。

当两个相近的数相减时，会引起有效数字的位数大大减少，误差增大。为了避免这种结 果，通常可以改变计算方法，将算式转化成等价的另一个计算公式。例如：

$$\sqrt{x+1}-\sqrt{x}=\frac{1}{\sqrt{x+1}+\sqrt{x}}，当 x 很大时$$

$$1-\cos x = 2\sin^2\frac{x}{2}，当 x 接近于 0 时$$

在除法运算中，应当避免除数接近于零，或者除数的绝对值远远小于被除数的绝对值的

情形，因为这两种情形都会使舍入误差增大，甚至使结果溢出。解决办法仍然是转化为等价算式。例如：

$$\frac{1}{\sqrt{x+1}-\sqrt{x}}=\sqrt{x+1}+\sqrt{x}，当 x 很大时$$

这里，不同计算公式的选择就如同上述不同计算次序的选择，虽然在数学上结果是一样 的，但在计算机中却存在很大差别。

计算量

站在计算机的角度，对数值方法主要关注的是算法的效率和精度。算法的效率由算法复 杂度决定，数值方法中通常用浮点乘除运算（flop）的次数来度量算法效率，称为算法的计 算量。计算量越小，效率就越高。

当一个算法的计算量很大，并不意味着它能提高计算结果的准确度，相反倒有可能使舍 入误差积累得更多，可谓费力不讨好。利用数学推导来简化计算公式，或者利用计算机的运 算及存贮能力来巧妙安排计算步骤，都可以减少计算量，使计算更快、更准确。

例如，设 A、B、C 分别是 10×20、20×50、50×1 的矩阵，我们来考虑如何计算 ABC。 一种算法是先算 AB，再乘 C，计算量为 10500flops；另一种算法是先算 BC，再用 A 乘， 计算量为 1200flops。显然后一种算法大大优于前一算法，再次显示了“次序”的妙处。

又如，考虑如何计算 x64。一种算法是将 64 个 x 逐步相乘，计算量为 63flops；另一算 法利用

$$x^{64} = x \cdot x^2 \cdot x^4 \cdot x^8 \cdot x^{16} \cdot x^{32},$$

其中 x**2k（k＝2，4，8，16）的计算都可以利用前一步算出的结果，即

$$x^{2k} = x^k \cdot x^k$$

这样计算量可以降至 10flops。 有些数值算法甚至会使计算量大到失去实际意义的地步，就如 Hanoi 塔问题的算法对较

大问题规模不可行一样。例如求解 n 元线性方程组的克莱默法则对于较大 n 就是不可行的方法，因为其计算量是(n+1)(n-1)(n!)+n；而高斯消去法的计算量仅为 n3/3+n2-n/3，是非常高效的算法。

病态与良态问题

有些问题的解对初始数据非常敏感，数据的微小变化会导致计算结果的剧烈变化，这种

问题称为病态问题，反之称为良态问题。例如多项式 p(x) = x2+x-1150 在 100/3 和 33 处的值分别为-5.6 和-28，数据变化只有 1%，而结果变化了 400%。又如下面这个方程组

$$\begin{cases} x_1 + \frac{1}{2}x_2 + \frac{1}{3}x_3 = \frac{11}{6} \\ \frac{1}{2}x_1 + \frac{1}{3}x_2 + \frac{1}{4}x_3 = \frac{13}{12} \\ \frac{1}{3}x_1 + \frac{1}{4}x_2 + \frac{1}{5}x_3 = \frac{47}{60} \end{cases}$$

的解是 x1 ＝ x2 ＝ x3 ＝1，当将各个系数舍入成两位有效数字，与原来的系数虽然差别不大，但方程组的解却变成了 x1 ≈ -6.22，x2 ＝38.25，x3 ＝ -33.65。

相反，下面这个方程组

$$\begin{cases} x_1 + 2x_2 = -2 \\ 2x_1 - x_2 = 6 \end{cases}$$

的解为 x1 =2，x2 = -2。若对其常数项-2 做微小扰动改为-2.005，则解变成 1.999 和-2.002， 与原来的解差别很小。可见这个问题是良态的。

数值方法主要研究良态问题的数值解法。由于实际问题的数据往往是近似值，或者是经 过舍入处理的，这相当于对原始数据的扰动，如果求解的是病态问题，则会导致很隐蔽的错 误结果。病态问题在函数计算、方程求根、方程组求解中都存在，它的计算或求解应当使用 专门

的方法，或者转化为良态问题来解决。

数值稳定性

求解一个问题的数值方法往往涉及大量运算，每一步运算一般都会产生舍入误差，前面 运算的误差也可能影响后面的运算。一个数值方法如果在计算过程中能将舍入误差控制在一 定范围内，就称为数值稳定的，否则称为数值不稳定的。例如，考虑下面这个积分的计算：

$$
\begin{aligned}
I_n &= \int_0^1 \frac{x^n}{x+5}dx \\
&= \int_0^1 \frac{x^n + 5x^{n-1} - 5x^{n-1}}{x+5} \\
&= \int_0^1 x^{n-1}dx - 5\int_0^1 \frac{x^{n-1}}{x+5}dx \\
&= \frac{1}{n} - 5I_{n-1}
\end{aligned}
$$

根据上面这个递推式，可得出迭代算法：

$$I_0 = \int_0^1 \frac{1}{x+5}dx = \ln 6 - \ln 5 \approx 0.1823$$

$$I_n = -5I_{n-1} + \frac{1}{n}$$

这个算法是不稳定的，因为 I0 的舍入误差会随着迭代过程不断传播、放大。编程计算一下

可见，结果中甚至出现了负数，而根据原积分式可知 In 应该总是大于 0。

```
>>> def f():
        x = 0.1823
    print "I0 =",x
    for n in range(1,101):
        x = -5 * x + 1.0 / n
        print "I"+str(n)+" =",x
>>> f()
I0 = 0.1823
I1 = 0.0885
I2 = 0.0575
I3 = 0.0458333333333
...
I97 = 1.36042495942e+63
I98 = -6.80212479709e+63
I99 = 3.40106239854e+64
I100 = -1.70053119927e+65
```

现在利用下列关系式

$$\frac{1}{6(n+1)} < I_n < \frac{1}{5(n+1)},$$

先对足够大的 n 取 I_n 的估计值，然后再计算 I_{n-1}、I_{n-2}、…、I_1。迭代算法如下：

$$I_{100} \approx \frac{1}{2}(\frac{1}{606}+\frac{1}{505}) \approx 0.001815$$

$$I_{n-1} = -\frac{1}{5}I_n + \frac{1}{5n},$$

这个算法可使误差逐渐减小，因此是数值稳定的，下面程序的运行结果验证了这一点。此例又一次显示了次序的重要性。

```
>>> def g():
        x = 0.001815
        print "I100 =",x
        for n in range(100,0,-1):
            x = -x/5 + 1.0/(5*n)
            print "I"+str(n-1)+" =",x
>>> g()
I100 = 0.001815
I99 = 0.001637
I98 = 0.0016928020202
I97 = 0.00170225592249
...
I3 = 0.043138734089
I2 = 0.0580389198489
I1 = 0.0883922160302
I0 = 0.182321556794
```

综上所述，数值方法以利用计算机进行数值计算的方式来解决科学和工程中抽象出来的 数学问题。与纯数学方法不同，数值计算方法的构造和算法实现必须考虑计算机的能力和限 制，亦即计算思维的原则对计算方法具有重要影响。

11.2 生物信息学

计算生物学（computational biology）研究如何用计算机来解决生物学问题，主要研究内 容包括对生物系统的数学建模、对生物数据的分析、模拟等。本节介绍计算生物学的一个分 支——生物信息学①。

生物信息学（bioinformatics）主要研究生物信息的存储、获取和分析，这里所说的生物 信息主要是指基因组信息。近年来，通过庞大的项目合作，生物学家对人类基因组和其他生 物的基因组进行测序，获得了大量的数据。针对以指数方式增长的数据，生物信息学应用算 法、数据库、机器学习等技术，来解决 DNA 和蛋白质序列的分析、序列分类、基因在序列 中的定位、不同序列的比对、蛋白质结构及功能的预测和新药物新疗法的发现等问题。生物 信息学已成为处于生命科学和计算机科学前沿的一门有战略意义的学科，对医学、生物技术 以及社会的许多领域都有重要影响。

生物信息的表示 为了利用计算机来处理生物信息，首先要将生物信息表示成计算机中的数据。例如，听上去很复杂的 DNA 和蛋白质的链状分子，出乎意料地很容易表示——用符号序列即可。

DNA 是由 4 种单体，即以 A（腺嘌呤）、C（胞嘧啶）、G（鸟嘌呤）、T（胸腺嘧啶）代 表的 4 中核苷酸聚合成的生物大分子。蛋白质是另一类由 20 种单体，即以 A、C、D、W 等 表示的 20 种氨基酸聚合成的大分子。在链状分子的特定位置上，只能出现某种确定的单体（“字符”），而不是几种可能字符的组合，因此分子链可以用一维的、不分岔的。有方向的字符序列来表示。例如，DNA 分子可表示成如“AGTGATG”一样的字符序列。

测定 DNA 和蛋白质链状分子的字符序列是从微观结构研究生物的出发点。 除了序列数据，生物信息还包括结构和功能数据、基因表达数据、生化反应通路数据、表现型和临床数据等。

生物信息数据库

数据库技术是管理大量数据的计算机技术，目的是使用户能够方便、高效地访问大量数据。过去数十年间，随着人类基因组测序工程和其他生物测序项目的完成或推进，以及诸如 DNA 微阵列等高效实验技术的出现，产生并积累了大量的生物信息（如前面所说的核苷酸 序列和氨基酸序列），因此需要利用数据库技术将这些信息组织、存储起来。有了生物信息 数据库，生物学家们通过易用的 GUI 来访问数据库，既可以读取数据，也可以添加新数据 或者修订老数据。当然，更重要的工作是利用各种算法来处理数据库中的生物数据。生物学 未来的新发现很可能是通过分析数据库中的生物数据获得的，而非仅仅依赖于传统的实验。

> ① 也有说生物信息学和计算生物学是一回事的。

互联网上有很多生物数据库，例如 EMBL（核苷酸序列数据库）、GenBank（基因序列 数据库）、PDB（蛋白质数据库）等等。

生物数据分析

建立了生物信息数据库之后，生物学家接下来的研究重点就转向了数据分析。庞大的生 物信息数据库对数据分析技术提出了具有挑战性的问题，人工分析 DNA 序列早已成为不可 能完成的任务，传统的计算机算法也越来越显示出不足，这促使生物信息学去寻求新的算法 来解决问题。

序列分析是生物信息学的主要研究内容。例如，通过分析数据库中的成千上万种有机体 的 DNA 序列，可以识别特定序列的结构和功能、特定序列在不同物种之间的不同形式、相 同物种内部特定序列的不同形式。又如，通过对一组序列进行比较，可以发现功能之间的相 似性或者物种之间的联系。还可以在一个基因组中搜索蛋白质编码基因、RNA 基因和其他 功能序列，可以利用 DNA 序列来识别蛋白质。

下面介绍基因组比对的基本思想和方法。当生物学家通过实验获得了一个基因序列，他 接着就要确定这个基因序列的功能。为此，他以这个基因序列作为输入，到基因序列数据库 中去搜索与之相似的、已知功能的基因序列，因为生物学家认为基因序列相似意味着功能相 似。一种衡量基因序列相似性的方法是基因组比对（genome alignment），该方法将两个基 因序列对齐（如果序列长度不同可以在序列中插入一些空白位置），然后为对齐的每一对（代 表核苷酸的）字符打分，所有分数的总和就是两个序列的相似度。例如，对于两个基因序列 AGTGATG 和 GTTAG，适当插入空白（用下划线字符“_”表示）后可以按如下方式对准：

```
A G T G A T G
_ G T T A _ G
```

假如按如下规则打分：

	A	C	G	T	_
A	5	-1	-2	-1	-3
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-2
T	-1	-2	-2	5	-1
_	-3	-4	-2	-1	

则该对准方案的得分为 14。当然也可以按别的方式对准，但上面给出的对准方案是得分最高的。这个最优对准方案可以利用动态规划算法求得。 另外，计算机科学中最新的机器学习和数据挖掘技术能够实现更复杂的数据分析，很自然地成为当今生物信息学所倚重的方法。机器学习和数据挖掘的领域界线并不明显，它们都 是关于从大量数据中发现知识、模式、规则的技术。具体技术包括神经网络、隐马尔可夫模 型、支持向量机、聚类分析等，这些技术都非常适合生物信息的分析和处理。例如，对大量 蛋白质序列进行聚类分析，可以将所有蛋白质序列分组，使得同组的蛋白质序列非常相似， 而不同组的蛋白质非常不相似。

11.3 计算物理学

计算物理学（computational physics）研究利用计算机来解决物理问题，是计算机科学、 计算数学和物理学相结合而形成的交叉学科。如今，计算物理已经与理论物理、实验物理一 起构成了物理学的三大支柱。

物理学旨在发现、解释和预测宇宙运行规律，而为了更准确地做到这一点，今天的物理 学越来越依赖于计算。首先，很多物理问题涉及海量的实验数据，依靠手工处理根本无力解决。例如在高能物理实验中，由于实验技术的发展和测量精度的提高，实验规模越来越大， 实验数据也大幅增加，只能利用计算机来处理实验数据。其次，很多物理问题涉及复杂的计 算，解析方法或手工数值计算无法解决这样的计算问题。例如电子反常磁矩修正的计算，对 四阶修正的手工解析技术已经相当繁杂，而对六阶修正的计算已经包含了 72 个费曼图，手 工解析运算已不可能完成。同样只能利用计算机来解决问题。

在物理学中运用计算思维，使我们可以利用数值计算、符号计算和模拟等方法来发现和 预测物理系统的特性和规律。

解决物理问题时，通常在获得描述物理过程的数学公式后，需要进行数值分析以便与实 验结果进行对照。对于复杂的计算，手工数值分析是不可能的，只能采用数值方法利用计算 机来计算。

有些物理问题不是数值计算问题，需要利用计算机的符号处理能力来解决。例如，理论 物理中的公式推导，就是纯粹的符号变换。有时即使是数值计算问题，由于精度要求很高， 导致计算耗时很长甚至无法达到所需精度，这时可以利用符号计算来推导出解析形式的问题 解。又如，有时数值方法是病态的，如果能将数值计算改成解析计算，则可以得到有意义的 结果。

统计物理中有个自回避随机迁移问题，它是在随机漫步中加上了一个限制，即以后的步 子不能穿过以前各步所走过的路径。这样的问题不像一般的迁移问题那样可以用微分方程来 描写系统的统计行为，计算机模拟几乎是唯一的研究方法。计算机模拟不受实验条件、时间 和空间的限制，只要建立了模型，就能进行模拟实验，因而具有极大的灵活性。下面通过一 个实例来介绍模拟方法在计算物理学中的应用。

热平衡系统的模拟

为了研究一个包含 N 个粒子的热系统，原则上只要了解每个粒子的运动，就能弄清楚 粒子和粒子之间的每一次相互作用。但由于粒子数目太大，要想计算 N 个粒子的轨迹以及 N(N-1)对相互作用，是非常困难的。

然而，对于处于平衡态的热系统，虽然系统的微观特性总是在变动，但其宏观特性则是 恒定不变的，体现为具有恒定的温度。系统的微观状态由每个粒子的速度等物理量来刻划， 粒子间的相互作用会导致微观状态改变；而系统的宏观状态是微观状态的集体特性，表现为 系统

的总能量（或温度等）。统计物理学认为，虽然微观状态可能没有规则，但宏观状态服 从统计规律。对于处于平衡态的理想气体而言，虽然微观相互作用可导致粒子能量的重新分 配，但系统的总能量保持不变。

考虑一个由三个粒子组成的小系统 S。假设共有 4 份能量在这三个粒子之间交换，则能 量分布可以有以下 15 种状态：(4,0,0)、(0,4,0)、(0,0,4)、(3,1,0)、(3,0,1)、(1,3,0)、(0,3,1)、(1,0,3)、(0,1,3)、(2,2,0)、(2,0,2)、(2,1,1)、(0,2,2)、(1,2,1)、(1,1,2)。这里元组(a,b,c)表示三个粒子各

自获得的能量。每种微观状态都有自己的出现概率，例如从这 15 种微观状态可见，一个粒子占有全部能量的概率为 3/15 = 0.2。S 的平衡特性由概率较高的微观状态决定，而通过随机抽样方法（蒙特卡洛方法）可以有效地产生高可能性微观状态，从而可以用来评估 S 的 平衡特性。

我们引入一个“demon”来与系统 S 发生相互作用。作用方式是：令 demon 与 S 中某 个随机选择的粒子进行相互作用，并试着随机改变该粒子的状态（对气体来说就是改变粒子 的速度）；如果这个改变导致粒子能量减少，则执行这个改变，并将少掉的能量传递给 demon；如果这个改变导致粒子能量增加，则仅当 demon 有足够能量传递给粒子时才执行这次改变。按这种方式，每次产生新的微观状态时，系统 S 的能量加上 demon 的能量保持不变。

具体地，将 demon 加入到 S（包含三个微观粒子）中后，宏观状态仍为 4 份能量。新系统“S+demon”的 demon 为 0 能量的状态共有 15 个，正对应于原始系统 S 的那 15 个状态。如果尝试改变一个微观状态使得某个粒子减少一份能量，则将那份能量转给 demon，这样就使原始系统变成了具有 3 份能量的系统，而 demon 具有 1 份能量。与这种情况对应的微观状态有 10 个，即(3,0,0)、(0,3,0)、(0,0,3)、(2,1,0)、(2,0,1)、(1,2,0)、(0,2,1)、(1,0,2)、(0,1,2)和(1,1,1)。由此可见，如果实施一系列的微观状态随机改变，将发现 demon 具有 1 份能量与 具有 0 份能量的相对概率为 10/15 = 2/3。也就是说，当 demon 扰乱小系统 S 时，S 仍然处于 原来的宏观能量的可能性更大，而不是处于某个较低能量。

同理，如果 demon 具有 2 份能量，则 S+demon 系统具有 6 个微观状态；如果 demon 具 有 3 份能量，则组合系统具有 3 种微观状态；如果 demon 拥有全部 4 份能量，则组合系统 只有一种微观状态。这几种情形对应的相对概率分别为 6/15、3/15 和 1/15。

一般地，对于一个宏观系统，当产生大量的微观状态改变之后，其中 demon 拥有能量 E 的微观状态，与 demon 拥有 0 能量的微观状态数目之比是随 E 的升高而呈指数形式下降的， 具体公式为

$$\frac{p(E_d = E)}{p(E_d = 0)} = e^{-E/kT}$$

其中 k 是玻尔兹曼常数，T 是宏观系统的温度。以我们的小系统 S 为例，p(Ed=1) / p(Ed = 0)约为 2/3。

总之，计算物理学依据理论物理提供的物理原理和数学方程，针对实验物理提供的实验 数据，进行数值计算或符号计算，从而为理论研究提供数据、帮助分析实验数据和模拟物理 系统。

11.4 计算化学

化学在传统上一直被认为是一门实验科学，但随着计算机技术的应用，化学家成为大规 模使用计算机的用户，化学科学的研究内容、方法乃至学科的结构和性质随之发生了深刻变 化。计算化学（computational chemistry）是化学和计算机科学等学科相结合而形成的交叉学科，其研究内容是如何利用计算机来解决化学问题。计算化学这个术语早在 1970 年就出现

了，并且在上世纪 70 年代逐步形成了计算化学学科。 因此，计算化学可以帮助实验化学家，或者挑战实验化学家来找出全新的化学对象。

有些化学问题是无法用分析方法解决的，只能通过计算来解决。计算化学一般用于解决 数学方法足够成熟从而能在计算机上实现的问题。计算化学有两个用途：一是通过计算来与 化学实验互为印证、互为补充；一是通过计算来预测迄今完全未知的分子或从未观察到的化 学现象，或者探索利用实验方法不能很好研究的反应机制。

计算化学的研究内容很多，我们简单介绍化学数据库和分子模拟（或分子建模），前者 是关于化学信息表示、存储和查找的，后者是研究化学系统结构和运动的。

化学数据库是专门存储化学信息的数据库，其中的化学信息可以是化学结构、晶体结构、 光谱、反应与合成、热物理等类型的数据。以化学结构数据为例，学过中学化学课程的人都 知道，化学家通常用直线表示原子之间的化学键，利用化学键将若干原子连接在一起，形成 了分子结构的二维表示。这种表示对化学家来说是理想的、可视的，但对化学数据的计算机 处理来说是很不合适的，尤其是对数据的存储和查找。为此，需要建立分子结构的计算机表示，如小分子可以用原子的列表或 XML 元素表示，而大分子（如蛋白质）可用氨基酸序列 来表示。当今一些大的化学结构数据库存储了成百万的分子结构数据（存储量高达 TB 级），可以方便而高效地查找信息。

分子模拟利用计算机程序来模拟化学系统的微观结构和运动，并用数值计算、统计方法 等对系统的热力学、动力学等性质进行理论预测。宏观化学现象是无数个分子（原子）的集 体行为，一般通过统计方法来研究。然而，化学统计力学通常仅适用于“理想系统”（如理 想气体、完美晶体等），量子力学方法也不适用于动力学过程和有温度压力变化的系统。作 为替代方法，分子模拟将原子、分子按经典粒子处理，提供了化学系统的微观结构、运动过 程以及与宏观性质相关的数据和直观图象，从而能在更一般的情形下研究系统行为。分子模 拟有两种主要方法，一是基于粒子运动的经典轨迹的分子动力学方法，一种是基于统计力学 的蒙特卡洛方法。分子模拟技术不仅在计算化学中有用，而且还可用于药物设计和计算生物 学中的分子系统（从小的化学系统到大的生物分子）。

计算化学内部还包括量子化学计算、化学人工智能、化学 CAD 和 CAI 等领域，可以解 决识别化学结构与性质之间的相关性、化合物的有效合成、设计能与其他分子按特定方式进 行反应的分子（如新药设计）等问题。解决问题过程中所用到的计算化学方法有些是高度精 确的，更多的则是近似的。计算化学的目标是使计算误差极小化，同时保证计算是可行的。

11.5 计算经济学

计算经济学（computational economics）是计算机科学与经济和管理科学相结合而形成 的交叉学科，其主要研究领域包括经济系统的计算模型、计算计量经济学、计算金融学等， 目的是利用计算技术和数值方法来解决传统方法无法解决的问题。这里，我们特别考虑建模 问题，简单介绍基于代理的计算经济学。

基于代理的（agent-based）模型是用于模拟自治个体的行为和相互作用的计算模型，目 的是从整个系统的层面来评估这些个体相互作用所产生的效果。基于代理的计算经济学（ACE）将经济过程建模为一个由相互作用的代理所构成的动态系统，并应用数值方法来模 拟系统的运行。ACE 中的“代理”是指按照一定的规则行事、并且相互作用的对象，可以 表示个体（如一个人）、社会群体（如一家公司）、生物体（如农作物）或物理系统（如交通 系统）。建模者要做的事情是为由多个相互作用的代理组成的系统提供初始条件，然后就不 加干涉地观察系统如何随时间而演化。系统中的代理完全通过相互作用来驱动系统向前发 展，没有任何外部强加的平衡条件。

ACE 方法的一个应用领域是资产定价。计算模型涉及许多代理，每个代理可以从一组 预测策略中选择特定策略去行事，比如预测股票价格。根据预测的结果，会影响代理们的资 产需求，而这又会影响股票价格。通过对模型的分析，可以获得有用的结果，例如，当代理 改变预测策略时，经常会引发资产价格的大波动。又如，有经济学家认为 ACE 可能对理解 最近的金融危机也是有用的方法。

总之，计算机科学的建模技术为计算经济学提供了非常有用的方法和工具。

11.6 练习

1. 举例说明计算机在你的专业领域中的应用。

2. 利用本书中学到的知识去解决一个你专业领域的问题。

3. 假如你是 X 专业的，现在有“计算 X 学”吗？如果有，该学科的研究内容是什么？

附录

1 Python 异常处理参考

本节简单罗列 Python 语言中与异常处理有关的常用语句形式及用法。 发生错误时通常由系统自动抛出异常，但也可由程序自己抛出并捕获。

捕获并处理异常：try-except

发生错误时，如果应用程序没有预定义的处理代码，则由 Python 的缺省异常处理机制 来处理，处理动作是中止应用程序并显示错误信息。如果程序自己处理异常，可编写 try-except 语句来定义异常处理代码。详见前面各节。

手动抛出异常：raise 异常可以由系统自动抛出，也可以由我们自己的程序手动抛出。Python 提供 raise 语句

用于手动抛出异常。下面的语句抛出一个 ValueError 异常，该异常被 Python 的缺省异常处理程序捕获：

```
>>> raise ValueError
Traceback (most recent call last): File "&lt;stdin&gt;", line 1, in &lt;module&gt;
ValueError
```

除了错误类型，raise 语句还可以带有错误的描述信息：

```
>>> raise ValueError, "Wrong value!"
Traceback (most recent call last): File "&lt;stdin&gt;", line 1, in &lt;module&gt;
ValueError: Wrong value!
```

当然也可以由程序自己处理自己抛出的异常，例如

```
>>> try:
        raise ValueError
    except ValueError:
        print "Exception caught!"
Exception caught!
```

用户自定义异常

前面程序例子中抛出的都是 Python 的内建异常，我们也可以定义自己的异常类型。为 此目的，需要了解 Python 的异常类 Exception 以及类、子类、继承等面向对象程序设计概念， 这些概念将在第 x 章中介绍。这里我们用下面的简单例子演示大致用法，以使读者先获得一 个初步印象：

```
>>> class MyException(Exception):
        pass
```

这是一个类定义，它在 Python 内建的 Exception 类的基础上定义了我们自己的异常类 MyException。虽然语句 pass 表明我们并没有在 Exception 类的基础上添加任何东西，但 MyException 确实是一个新的异常类，完全可以像 Python 内建的各种异常一样进行抛出、捕获。例如：

```
>>> try:
        raise MyException
    except MyException:
        print "MyException caught!"
MyException caught!
```

确保执行的代码：try-finally

一般来说，发生异常之后，控制都转到异常处理代码，而正常算法部分的代码不再执行。

Python 的异常处理还允许我们用 try-finally 语句来指定这样的代码：不管是否发生异常，这些代码都必须执行。这种机制可以用来完成出错后的扫尾工作。例如：

```
>>> try:
        x = input("Enter a number: ")
        print x
    finally:
        print "This is final!"
Enter a number: 123
123
This is final!
```

本例中，我们为 x 输入了一个正常数值 123，故 try 语句块没有发生异常，显示 123 后 又执行了最后的 print 语句。为什么不写成如下形式呢？

```
x = input("Enter a number: ")
print x
print "This is final!"
```

区别在于，当发生错误时，这种写法就有可能未执行最后的 print 语句，而 try-finally 的写法则在发生异常的情况下也会确保执行最后的 print 语句。例如我们再次执行上面的语 句：

```
>>> try:
        x = input("Enter a number: ")
        print x
    finally:
        print "This is final!"
Enter a number: abc
This is final!
Traceback (most recent call last): File "&lt;stdin&gt;", line 2, in &lt;module&gt; File "
NameError: name 'abc' is not defined
```

可见，由于输入数据错误，导致 try 语句块发生异常而无法继续，但 finally 下面的 语句却得到了执行。仅当 finally 部分确保执行之后，控制才转到（缺省）异常处理程序 来处理捕获到的异常。

一般形式：try-except-finally

这种形式的异常处理语句综合了 try-except 和 try-finally 的功能。首先执行 try 部分，如 果一切正常，再执行 finally 部分。try 部分如果出错，则还是要执行 finally 部分，然后再由 except 部分来处理异常。

2 Tkinter 画布方法

本节罗列 Canvas 对象的方法，供需要的读者编程时参考。具体用法请查阅参考资料。

创建图形项的方法

- create_arc(<限定框>, <选项>)：创建弧形，返回标识号
- create_bitmap(<位置>, <选项>)：创建位图，返回标识号
- create_image(<位置>, <选项>)：创建图像，返回标识号
- create_line(<坐标序列>, <选项>)：创建线条，返回标识号
- create_oval(<限定框>, <选项>)：创建椭圆形，返回标识号
- create_polygon(<坐标序列>, <选项>)：创建多边形，返回标识号
- create_rectangle(<限定框>, <选项>)：创建矩形，返回标识号
- create_text(<位置>, <选项>)：创建文本，返回标识号
- create_window(<位置>, <选项>)：创建窗口型构件，返回标识号

操作画布上图形项的方法

- delete(<图形项>)：删除图形项
- itemcget(<图形项>, <选项>)：获取某图形项的选项值
- itemconfig(<图形项>, <选项>)：设置图形项的选项值
- itemconfigure(<图形项>, <选项>)：同上
- coords(<图形项>)：返回图形项的坐标
- coords(<图形项>, x0, y0, x1, y1, ..., xn, yn)：改变图形项的坐标
- bbox(<图形项>)：返回图形项的界限框（坐标）
- bbox()：返回所有图形项的界限框
- canvasx(<窗口坐标 x>)：将窗口坐标 x 转换成画布坐标 x
- canvasy(<窗口坐标 y>)：将窗口坐标 y 转换成画布坐标 y
- type(<图形项>)：返回图形项的类型
- lift(<图形项>)：将图形项移至画布最上层
- tkraise(<图形项>)：同上
- lower(<图形项>)：将图形项移至画布最底层
- move(<图形项>, dx, dy)：将图形项向右移动 dx 单位，向下移动 dy 单位
- scale(<图形项>, <x 比例>, <y 比例>, <x 位移>, <y 位移>)：根据比例缩放图形项

查找画布上图形项的方法

下列方法用于查找某些项目组。对每个方法，都有对应的 addtag 方法。不是处理 find 方法返回的每个项目，而是为一组项目增加一个临时标签、一次性处理所有具有该标签的项 目、然后删除该标签，常常可以得到更好的性能。

- find_above(<图形项>)：返回位于给定图形项之上的图形项

- find_all() ：返回画布上所有图形项的标识号构成的元组，等于 find_withtag(ALL)
- find_below(<图形项>)：返回位于给定图形项之下的图形项
- find_closest(x, y)：返回与给定位置最近的图形项，位置以画布坐标给出
- find_enclosed(x1, y1, x2, y2)：返回被给定矩形包围的所有图形项
- find_overlapping(x1, y1, x2, y2)：返回与给定矩形重叠的所有图形项
- find_withtag(<图形项>)：返回与给定标识匹配的所有图形项

操作标签的方法

- addtag_above(<新标签>, <图形项>)：为位于给定图形项之上的图形项添加新标签
- addtag_all(<新标签>)：为画布上所有图形项添加新标签，即 addtag_withtag(<新 标签>, ALL)
- addtag_below(<新标签>, <图形项>)：为位于给定图形项之下的图形项添加新标签
- addtag_closest(<新标签>, x, y)：为与给定坐标最近的图形项添加新标签
- addtag_enclosed(<新标签>, x1, y1, x2, y2)：为被给定矩形包围的所有图形项添 加新标签
- addtag_overlapping(<新标签>, x1, y1, x2, y2) ：为与给定矩形重叠的所有图 形项添加新标签
- addtag_withtag(<新标签>, <标签>)：为具有给定标签的所有图形项添加新标签
- dtag(<图形项>, <标签>)：为给定图形项删除给定标签
- gettags(<图形项>：返回与给定图形项关联的所有标签

3 Tkinter 编程参考

3.1 构件属性值的设置

Tkinter 构件对象有很多属性，这些属性的值可以在创建实例时用关键字参数指定（未 指定值的属性都有缺省值）：

```
<构件类>(<父构件>,<属性>=<值>,...)
```

也可以在创建对象之后的任何时候通过调用对象的 configure（或简写为 config）方法来更改属性值：

```
<构件实例>.config(<属性>=<值>,...)
```

构件类还实现了一个字典接口，可使用下列语法来设置和查询属性：

```
<构件实例>["<属性>"] = <值> value = <构件实例>["<属性>"]
```

由于每个赋值语句都导致对 Tk 的一次调用，因此若想改变多个属性的值，较好的做法 是用 config 一次性赋值。

有些构件类的属性名称恰好是 Python 语言的保留字（如 class、from 等），当用关键词 参数形式为其赋值时，需要在选项名称后面加一个下划线（如 class、*from*等）。

3.2 构件的标准属性

Tkinter 为所有构件提供了一套标准属性，用来设置构件的外观（大小、颜色、字体等） 和行为。

设置构件的长度、宽度等属性时可选用不同的单位。缺省单位是像素，其他单位包括 c（厘米）、i（英寸）、m（毫米）和 p（磅，约 1/72 英寸）。

颜色

多数构件具有 background（可简写为 bg）和 foreground（可简写为 fg）属性，分别用于 指定构件的背景色和前景（文本）色。颜色可用颜色名称或红绿蓝（RGB）分量来定义。

所有平台都支持的常见颜色名称有"white"、"black"、"red"、"green"、"blue"、"cyan"、"yellow"、"magenta"等，其他颜色如 LightBlue、Moccasin、PeachPuff 等等也许依赖于具体的安装平台。颜色名称不区分大小写。大多数复合词组成的颜色名称也可以在使用单词间加 空格的形式，如"light blue"。

通过 RGB 分量值来指定颜色需使用特定格式的字符串："#RGB"、"#RRGGBB"、"#RRRGGGBBB"和"#RRRRGGGGBBBB"，它们分别用 1～4 个十六进制位来表示红绿蓝分量值，即分别将某颜色分量细化为 16、256、4096 和 65536 级。如果读者不熟悉十六进制，可以用下面这个方法将十进制数值转换成颜色格式字符串，其中宽度可选用 01～04：

```
my_color = "#%02x%02x%02x" % (128,192,200)
```

字体

多数构件具有 font 属性，用于指定文本的字体。一般情况下使用构件的缺省字体即可， 如果实在需要自己设置字体，最简单的方法是使用字体描述符。

字体描述符是一个三元组，包含字体族名称、尺寸（单位为磅）和字形修饰，其中尺寸 和字形修饰是可选的。当省略尺寸和字形修饰时，如果字体族名称不含空格，则可简单地用 字体族名称字符串作为字体描述符，否则必须用元组形式（名称后跟一个逗号）。例如下列 字体描述符都是合法的：

```
("Times",10,"bold") ("Helvetica",10,"bold italic") ("Symbol",8) ("MS Serif",) "Courier"
```

Windows 平台上常见的字体族有 Arial、Courier New（或 Courier）、Comic Sans MS、Fixedsys、Helvetica（同 Arial）、MS Sans Serif、MS Serif、Symbol、System、Times New Roman（或 Times）和 Verdana 等。字形修饰可以从 normal、bold、roman、italic、underline 和 overstrike 中选用一个或多个。

除了字体描述符，还可以创建字体对象，这需要导入 tkFont 模块，并用 Font 类来创建 字体对象。在此不详述。

边框

Tkinter 的所有构件都有边框，某些构件的边框在缺省情形下不可见。边框宽度用 borderwidth（可简写为 border 或 bd）设置，多数构件的缺省边框宽度是 1 或 2 个像素。可以用属性 relief 为边框设置 3D 效果，可用的 3D 效果有'flat'或 FLAT、'groove'或 GROOVE、'raised'或 RAISED、'ridge'或 RIDGE、'solid'或 SOLID、'sunken'或 SUNKEN（见图 8.28）。



图 8.28 按钮边框 3D 效果

文本

标签、按钮、勾选钮等构件都有 text 属性，用于指定有关的文本。文本通常是单行的， 但利用新行字符\n 可以实现多行文本。多行文本的对齐方式可以用 justify 选项设置，缺省 值是 CENTER，可用值还有 LEFT 或 RIGHT。

图像

很多构件都有 image 属性，用于显示图像。例如命令按钮上可以显示图像而不是文本，标签也可以是图像而非文本，Text 构件可以将文本和图像混合编辑。

image 属性需要一个图像对象作为属性值。图像对象可以用 PhotoImage 类来创建，图像 的来源可以是.gif 等格式的图像文件。

例如：

```
>>> root = Tk()
>>> img = PhotoImage(file="d:\mypic.gif")
>>> Button(root,image=img).pack()
```

3.3 各种构件的属性

除了标准属性，每种构件类还有独特的属性。这里仅以 Button 类为例列出按钮构件的 常用属性，其他构件类仅列出类名，具体有哪些属性请查阅 Tkinter 参考资料。

Button

构造器：`Button(parent, option = value, ... )`

常用选项：

- anchor：指定按钮文本在按钮中的位置（用方位值表示）。
- bd 或 borderwidth：按钮边框的宽度，缺省值为 2 个像素。
- bg 或 background：背景色。 command：点击按钮时调用的函数或方法。
- default：按钮的初始状态，缺省值为 NORMAL，可改为 DISABLED（不可用状态）。
- disabledforeground：不可用状态下的前景色。
- fg 或 foreground：前景色（即文本颜色）。
- font：按钮文本字体。 height：按钮高度（对普通按钮即文本行数）。
- justify：多行文本的对齐方式（LEFT，CENTER，RIGHT）。
- overrelief：当鼠标置于按钮之上时的 3D 风格，缺省为 RAISED。 padx：文本左右留的空白宽度。
- pady：文本上下留的空白宽度。 relief：按钮边框的 3D 风格，缺省值为 RAISED。
- state：设置按钮状态（NORMAL，ACTIVE，DISABLED）。
- takefocus：按钮通常可成为键盘焦点（按空格键即为点击），将此选项设置为 0 则不能成为 键盘焦点。
- text：按钮上显示的文本，可以包含换行字符以显示多行文本。
- textvariable：与按钮文本关联的变量（实为 StringVar 对象），用于控制按钮文本内容。
- underline：缺省值为-1，意思是按钮文本的字符都没有下划线；若设为非负整数，则对应, 位置的字符带下划线。
- width：按钮宽度（普通按钮以字符为单位）。

Checkbutton

Entry

Frame

Label

LabelFrame

Listbox

Menu

Menubutton

Message

OptionMenu

PanedWindow

Radiobutton

Scale

Scrollbar

Spinbox

Text

Toplevel

3.4 对话框

GUI 的一个重要组成部分是弹出式对话框，即在程序执行过程中弹出一个窗口，用于与 用户的特定交互。Tkinter 提供了若干种标准对话框，用于显示消息、选择文件、输入数据 和选择颜色。

tkMessageBox 模块

本模块定义了若干种简单的标准对话框和消息框，它们可通过调用以下函数来创建：askokcancel、askquestion、askretrycancel、askyesno、showerror、showinfo 和 showwarning。 这些函数的调用语法是：

```
function(title, message, options)
```

其中 title 设置窗口标题，message 设置消息内容（可用\n 显示多行消息），options 用于设置 各种选项。

这些函数的返回值依赖于用户所选择的按钮。函数 askokcancel、askretrycancel 和 askyesno 返回布尔值：True 表示选择了 OK 或 Yes，False 表示 No 或 Cancel。函数 askquestion 返回字符串 u"yes"或 u"no" ，分别表示选择了 Yes 和 No 按钮。

参数 options 可设置以下选项：

```
default = constant
```

指定缺省按钮。其中 constant 的值可以取 CANCEL、IGNORE、NO、OK、RETRY 或 YES。如果未指定，则第一个按钮（"OK"、"Yes"或"Retry"）将成为缺省按钮。

```
icon = constant
```

指定用什么图标。其中 constant 的值可以取 ERROR、INFO、QUESTION 或 WARNING。

```
parent = window
```

指定消息框的父窗口。如果未指定，则父窗口为根窗口。关闭消息框时，焦点返回到父 窗口。

tkFileDialog 模块

本模块定义了两种弹出式对话框，分别用于打开文件和保存文件的场合。通过调用函数 askopenfilename 和 asksaveasfilename 来创建所需对话框，调用语法是：

```
function(options)
```

如果用户选择了一个文件，则函数的返回值是所选文件的完整路径；如果用户选择了“取消”按钮，则返回一个空串。参数 options 可用的选项包括：

```
defaultextension = string
```

缺省文件扩展名。string 是以"."开头的字符串。

```
filetypes = [(filetype,pattern),...]
```

用若干个二元组来限定出现在对话框中的文件类型。每个二元组中的 filetype 指定文件 类型（即扩展名），pattern 指定文件名模式。这些信息将出现在对话框中的“文件类型”下 拉框中。

```
initialdir = dir
```

指定初始显示的目录路径。缺省值为当前工作目录。

```
initialfile = file
```

指定在“文件名”域初始显示的文件名。

```
parent = window
```

指定对话框的父窗口。缺省值为根窗口。

```
title = string
```

指定对话框窗口的标题。

tkSimpleDialog 模块

本模块用于从用户输入数据。通过调用函数 askinteger、askfloat 和 askstring 弹出输入对 话框。这些函数的调用语法是：

```
function(title, prompt, options)
```

其中 title 指定对话框窗口的标题，prompt 指定对话框中的提示信息，options 是一些选项。返回值是用户输入的数据。参数 options 可设置的一些选项包括：

```
initialvalue = value
```

指定对话框输入域中的初始值。

```
minvalue = value
```

指定合法输入的最小值。

```
maxvalue = value
```

指定合法输入的最大值。

tkColorChooser 模块

本模块提供选择颜色的对话框。通过调用函数 askcolor 即可弹出颜色对话框：

```
result = askcolor(color,options)
```

其中参数 color 指定显示的初始颜色，缺省值为淡灰色。参数 options 可设置的选项包括：

```
title = text
```

指定对话框窗口的标题，缺省为“颜色”。

```
parent = window
```

指定对话框的父窗口。缺省为根窗口。 如果用户点击“确定”按钮，返回值为元组(triple, color)，其中 triple 是包含红绿蓝分量的 三元组(R, G, B)，各分量值的范围是[0,255]，color 是所选颜色（Tkinter 颜色对象）。如果用 户点击“取消”按钮，则返回值为(None, None)。

3.5 事件

事件描述符是一个字符串，由修饰符、类型符和细节符三个部分构成：

```
<修饰符>-<类型符>-<细节符>
```

类型符

事件类型有很多，下面列出较常用的类型符：

Activate

构件从无效状态变成激活状态。

Button

用户点击鼠标按键。具体按键用细节符描述。

ButtonRelease

用户释放鼠标按键。在多数情况下用这个事件可能比 Button 更好，因为如果用户无意 点击了鼠标，可以将鼠标移开构件再释放，这样就不会触发该构件的点击事件。

Configure

用户改变了构件（主要是窗口）大小。

Deactivate

构件从激活状态变成无效状态。

Destroy

构件被撤销。

Enter

用户将鼠标指针移入构件的可见部分。

FocusIn

构件获得输入焦点。通过 Tab 键或 focus_set()方法可使构件获得焦点。

FocusOut

输入焦点从构件移出。

KeyPress

用户按下键盘上的某个键。可简写为 Key。具体按键用细节符描述。

KeyRelease

用户松开按键。

Leave

用户将鼠标指针移开构件。

Motion

用户移动鼠标指针。

修饰符

下面是常用的修饰符：

Alt

用户按下并保持 alt 键。

Control

用户按下并保持 control 键。

Double

在短时间内连续发生两次事件。例如<Double-Button-1>表示快速双击鼠标左键。

Shift

用户按下并保持 shift 键。

Triple

在短时间内连续发生三次事件。

细节符

鼠标事件的细节符用于描述具体绑定的是哪一个鼠标键，1、2、3 分别表示左、中、右 键。

键盘事件的细节符用于描述具体绑定的是哪一个键。对键的命名有多种方式，它们分别对应于 Event 对象中的如下几个属性：

char

如果按下 ASCII 字符键，此属性即是该字符；如果按下特殊键，此属性为空串。

keycode

键码，即所按键的编码。注意，键码未反映修饰符的情况，故无法区分该键上的不同字 符，即它不是键上字符的编码，故 a 和 A 具有相同的键码。

keysym

键符。如果按下普通 ASCII 字符键，键符即是该字符；如果按下特殊键，此属性设置 为该键的名称（是个字符串）。

keysym_num

键符码，是等价于 keysym 的一个数值编码。对普通单字符键来说，就是 ASCII 码。与 键码不同的是，键符码反映了修饰符的情况，因此 a 和 A 具有不同的键符码。

除了可打印字符，常见的特殊按键的键符包括：Alt_L，Alt_R，BackSpace，Cancel，Caps_Lock，Control_L，Control_R，Delete，Down，End，Escape，F1～F12，Home，Insert， Left，KP_0～KP_9，Next，Num_Lock，Pause，Print，Prior，Return，Right，Scroll_Lock， Shift_L，Shift_R，Tab，Up 等等。

常用事件

根据以上介绍的事件描述符的组成，可以构造如下常用事件：

- <Button-1>：左键点击
- <Button-2>：中键点击
- <Button-3>：右键点击
- <Double-Button-1>：左键双击
- <Triple-Button-1>：左键三击
- <B1-Motion>：左键按下并移动，每移一点都触发事件
- <B2-Motion>：中键按下并移动，每移一点都触发事件
- <B3-Motion>：右键按下并移动，每移一点都触发事件
- <ButtonRelease-1>：左键按下并释放
- <ButtonRelease-2>：中键按下并释放
- <ButtonRelease-3>：右键按下并释放
- <Enter>：进入按钮区域
- <Leave>：离开按钮区域
- <FocusIn>：键盘焦点移到构件或构件的子构件上
- <FocusOut>：键盘焦点从本构件移出 a：用户按下小写字母“a” 可打印字符（字母、数字和标点符号）都类似字母 a 这样使用。只有两个例外：空格键 对应的事件<space>，小于号对应的事件是<less>。
- <Shift-Up>：同时按下 Shift 键和↑键。
- 与<Shift-Up>类似的还有利用 Shift、Alt 和 Ctrl 构成的各种组合键，例如<Control-a>，
- <Control-Alt-a>等等。
- <Key>：按下任意键。
- 具体按下的键值由传递给回调函数的事件对象的 char 属性提供。如果是特殊键，char 属性值为空串。注意，如果输入上档键（如@#$%^&*之类），当按下 Shift 键时就触发了 <Key> 事件，再按下上档键又会触发<Key>。
- <Configure>：构件改变大小或位置。构件的新尺寸由事件对象的 width 和 height 属性传

递。

事件对象

每个事件都导致系统创建一个 Event 对象，该对象将被传递给事件处理程序，从而事件 处理函数能够从该对象的属性获得有关事件的各种信息。事件对象的属性包括：

x，y

鼠标点击位置坐标（相对于构件左上角），单位是像素。

x_root，y_root

鼠标点击位置坐标（相对于屏幕左上角），单位是像素。

num char

鼠标键编号，1、2、3 分别表示左、中、右键。

如果按下 ASCII 字符键，此属性即是该字符；如果按下特殊键，此属性为空串。

keycode

所按键的编码。注意，此编码无法区分该键上的不同字符，即它不是键上字符的编码。

keysym

如果按下普通 ASCII 字符键，此属性即是该字符；如果按下特殊键，此属性设置为该 键的名称（是个字符串）。

keysym_num：这是 keysym 的数值表示。对普通单字符键来说，就是 ASCII 码。

width，height

构件改变大小后的新尺寸（宽度和高度），单位是像素。仅适用于<Configure>事件。

widget

生成这个事件的构件实例。

参考文献

[1] Algorithmics, The Spirit of Computing, D. Harel, Y. Feldman，电子版。

[2] Computational Thinking, J. M. Wing, CACM, Vol. 49, No. 3, 2006。

[3] How to Think Like a Computer Scientist, Learning with Python，A. Downey，电子版。

[4] Learning Python, M. Lutz，电子版。

[5] Python Programming: An Introduction to Computer Science，J. Zell，电子版。

[6] Tkinter 8.4 reference: a GUI for Python, J. Shipman，电子版。

[7] http://en.wikipedia.org/

[8] Python 3 程序开发指南（第二版），M. Summerfield 著，王弘博等译，人民邮电出版社。

[9] 问题求解与编程概念（第 6 版），M. Sprankle 著，张晓明等译，清华大学出版社。

[10] 面向对象的思考过程（原书第二版），M. Weisfeld 著，杨会珍等译，中国水利水电出版社。

[11] 计算机算法与程序设计实践，董东、周丙寅编著，清华大学出版社。

[12] 计算方法教程，凌永祥，陈明逵，西安交通大学出版社。

[13] 生物信息学——机器学习方法，皮埃尔巴尔迪等著，张东辉译，中信出版社。

[14] 计算物理学，马文淦，中国科学技术大学出版社。

[15] 数字文明：物理学和计算机，郝柏林，张淑誉著，科学出版社。